Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anže Žagar

PRIKAZOVANJE MULTIMEDIJSKIH PREDSTAVITEV, PROJICIRANIH NA PLOSKVE NAVIDEZNE KOCKE

Diplomsko delo univerzitetnega študija

Mentor: prof. dr. Saša Divjak

Ljubljana, 2008

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

Zahvala

Najlepše se zahvaljujem vsem, ki so mi na tak ali drugačen način pomagali pri izdelavi tega diplomskega dela: prof. dr. Saši Divjaku za mentorstvo in usmerjanje, bratu Klemnu za vsebinski pregled, prof. Mateji Osredkar za lektorsko delo ter nenazadnje staršema, Slavku in Saši, ki sta mi tekom študija vedno stala ob strani in se po svojih močeh trudila, da mi pri tem omogočita najboljše pogoje.

Kazalo

Seznam uporabljenih kratic in simbolov

ACM Association for Computing Machinery

API Application Programming Interface

AWT Abstract Window Toolkit

BSD Berkeley Software Distribution

CAD Computer Aided Design

D3DX Direct3D Extension

Ecma (prvotno) European Computer Manufacturers Association

EPL Eclipse Public License

GPL General Public License

IDE Integrated Development Environment

IEC International Electrotechnical Commission

ISO International Organization for Standardization

JFC Java Foundation Classes

JNI Java Native Interface

JOGL Java OpenGL

JRE Java Runtime Environment

LWJGL Lightweight Java Game Library

MVC Model-View-Controller

OASIS Organization for the Advancement of Structured Information Standards

OpenGL Open Graphics Library

OpenGL ARB OpenGL Architecture Review Board

OSGi (prvotno) Open Services Gateway initiative

PDF Portable Document Format

RCP Rich Client Platform

SIGGRAPH Special Interest Group on Graphics and Interactive Techniques

SWF Shockwave Flash

SWT Standard Widget Toolkit

UML Unified Modeling Language

XML Extensible Markup Language

Povzetek

Cilj diplomskega dela je bila izvedba aplikacije za prikaz multimedijskih predstavitev. Aplikacija izstopa v 3-D učinkih, ki pri menjavi predstavitvenih strani dajejo vtis, kot da je predstavitev projicirana na ploskve navidezne kocke. Najprej sem opisal tematiko, predstavil sorodne rešitve ter pojasnil nekatere tehnologije, ki so bile na takšen ali drugačen način pomembne pri razvoju. V nadaljevanju sem se osredotočil na arhitekturo ter zasnovo aplikacije *jPresenter*, ki je bila v sklopu tega dela razvita v jeziku *Java*. Predstavil sem dve rešitvi za upodabljanje *PDF* dokumentov. Eno sem izvedel s pomočjo knjižnice *jPedal*, drugo pa preko vmesnika *JNI* s knjižnico *Poppler*. Nato sem opisal izvedeno 3-D grafiko, osnovano na knjižnici *JOGL*, ki je nekakšna ovojnica za knjižnico *OpenGL* v okolju *Java*. Na kratko sem razložil tudi nekatere uporabljene matematične prijeme pri animacijah in transformacijah v 3-D prostoru, kot so kvaternioni, Eulerjevi koti, Bézierove krivulje, B-zlepki ... V zaključku sem spregovoril še o prednostih in slabostih aplikacije *jPresenter* ter o možnostih za nadaljnje delo.

Ključne besede:

Multimedijske predstavitve, *Eclipse RPC*, *OpenGL*, *JOGL*, afine transformacije, kvaternioni, Bézierove krivulje, B-zlepki, de Casteljaujev algoritem, de Boor-Coxov algoritem.

Poglavje 1

Uvod

Predstavitev je proces predstavljanja neke snovi dani publiki. Poznamo jo bodisi kot neformalen govor ali pa zaobsega precej bolj kompleksne priprave, kjer si govorec lahko pomaga denimo z ročno pripravljenimi grafikoni ali pa morda računalniško podprtimi multimedijskimi prikazi.

V tem delu je poudarek na tehnični izvedbi predstavitev. Vsebinska plat in metode priprave predstavitev za doseganje optimalnih učinkov pri dani publiki presegajo obseg te diplomske naloge, zato bralcu, ki ga to zanima, svetujem branje druge literature, na primer [16]. Na kratko pa lahko povzamem, da sam način izvedbe predstavitve močno zavisi od njenega namena. Tako za bolj tehnične in znanstvene predstavitve, ki so namenjene predstavitvi znanj, tehnologij ali pa denimo rezultatov nekih raziskav, ne želimo pretiravati z estetiko. Dodatni učinki, ki s samo vsebino nimajo nobene povezave, lahko nehote odvračajo pozornost občinstva od bistva problema in povzročijo nižjo koncentracijo ter slabše razumevanje predstavljene snovi. Nasprotno pa ima vsesplošen vtis publike precej večji pomen, ko je cilj predstavitve bolj tržno usmerjen. Takrat si praviloma lahko v zmerni količini pomagamo tudi z učinki, ki so namenjeni zgolj povečevanju pozornosti in navduševanju publike. Bistvo te diplomske naloge je izvedba računalniške aplikacije, namenjene prikazovanju multimedijskih predstavitev, ki uporablja tovrstne učinke za izmenjavanje sicer statičnih predstavitvenih strani. Najprej pa bom na kratko opisal nekaj že obstoječih tehnologij, ki se dandanes najbolj pogosto uporabljajo na tem področju, in tehnologij, ki so bile tako ali drugače pomembne pri izdelavi omenjene aplikacije.

1.1 Motivacija

Ideja o izdelavi aplikacije za prikaz predstavitev se je deloma porodila iz potrebe. Podobni uporabniški programi sicer že obstajajo, a kot bomo videli v nadaljevanju (poglavje 1.2), zaradi takih ali drugačnih razlogov niso zadovoljili nekaterih mojih pričakovanj.

Po drugi strani mi je zanimiv izziv predstavljalo spoznavanje s 3-D grafiko, matematičnimi prijemi pri animacijah in transformacijah v 3-D prostoru ter nenazadnje tehnologijami kot so *OpenGL*, *Eclipse RCP*, *Java JNI* ...

Od aplikacije za prikaz predstavitev sem si v prvi vrsti želel rešitev, ki deluje tudi v okolju *Linux*. Izmed obstoječih tovrstnih uporabniških programov morda še najbolj izstopa *OpenOffice.org Impress*, ki pa za dinamične učinke uporablja le 2-D grafiko in tudi ta na mojem računalniku ne deluje povsem tekoče. Predstavo o tem, kako bi aplikacija za prikaz predstavitev lahko izgledala, mi je oblikovala tudi izkušnja s 3-D grafiko v upravitelju oken *Compiz*.

Compiz

V začetku leta 2006 je *Novell* predstavil prvo različico sestavljalnega upravitelja oken (angl. *compositing window manager*) za *X* okenski sistem, imenovan *Compiz* [21]. Ta upravljalnik oken je prevzel nekaj funkcionalnosti od operacijskega sistema *Apple OS X* (npr. *Exposé* za bolj pregledno preklapljanje med odprtimi okni) in dodal nekaj novih. Njegova morda najbolj značilna lastnost pa je kocka, ki ima na vertikalnih ploskvah projicirane štiri delovne površine. Izmenjava delovnih površin tako izgleda kot obračanje kocke, možna pa je tudi ročna navigacija kocke s pomočjo miške. Za hitro izrisovanje *Compiz* izkorišča podporo za strojno pospešeno grafiko z uporabo grafične knjižnice *OpenGL* (poglavje 1.4.3). Morda v kontekstu tega velja omeniti še projekt *Beryl*, ki se je začasno odcepil od razvoja *Compiz* projekta, a sta se marca 2007 oba projekta ponovno združila.

Omenjena kocka mi je služila kot ideja o tem, kako bi aplikacija za prikazovanje predstavitev lahko menjavala predstavitvene strani. Na spletu sem našel zelo preprost program, imenovan *PDF Cube* (poglavje 1.2), ki v osnovi počne prav to. Za predstavitvene strani pa uporabi posamezne strani poljubnega *PDF* dokumenta (poglavje 1.3). Njegova glavna pomanjkljivost je zelo skromna funkcionalnost.

Kasneje sem se seznanil še z nekaterimi drugimi rešitvami, ki so preprosto izvedbo menjave predstavitvenih strani z učinkom obračanja kocke nudile že precej prej. Te rešitve bom predstavil v nadaljevanju, ponovno pa nastane problem, ker so razvite za druge operacijske sisteme in so po večini plačljive.

1.2 Obstoječe rešitve

Ponudba uporabniških programov za delo z multimedijskimi predstavitvami obsega take, ki so namenjeni zgolj urejanju in pripravi predstavitvenih dokumentov, in take, ki zmorejo le njihovo prikazovanje ali pa združujejo obe funkcionalnosti. Namenjen izključno pripravi predstavitve je na primer *Beamer*, ki je razširitev urejevalnika tehničnih in znanstvenih besedil, *LaTeX*, in kot izhod pripravi *PDF* dokument. Pripravljen dokument lahko potem prikazujemo s poljubnim prikazovalnikom *PDF* dokumentov, denimo z že omenjenim *PDF Cube* ali pa *Adobe Acrobat*. Seveda za pripravo predstavitve v *PDF* obliki lahko uporabimo tudi množico drugih uporabniških programov, ki niso namenjeni izključno pripravi predstavitvenih dokumentov.

V nadaljevanju bom dal poudarek prikazovalnim lastnostim nekaterih najbolj znanih uporabniških programov, namenjenih delu z multimedijskmi predstavitvami.

Apple Keynote

Apple Keynote se je na trgu pojavil leta 2003 kot alternativa takrat že precej razširjenemu *Microsoft PowerPointu*. Od leta 2005 je del zbirke *Applovih* pisarniških orodij *iWork*.

Keynote že od leta 2003 v osnovnem paketu ponuja nekatere 3-D učinke, med drugim tudi že omenjen učinek obračanja kocke za prehajanje med predstavitvenimi stranmi. Pri tem izkorišča podporo za strojno pospešeno grafiko z uporabo grafične knjižnice *OpenGL*. Omogoča tako pripravo predstavitve kot tudi njeno prikazovanje. Ima svoj lasten datotečni zapis, a med drugim podpira tudi izvoz v *PDF* in *PowerPoint* (poglavje 1.3). Njegova glavna omejitev je, da deluje zgolj v operacijskem sistemu *Mac OS X*. *Keynote* je zaprtokoden in plačljiv.

Microsoft PowerPoint

PowerPoint je leta 1987 razvilo podjetje *Forethought* za okolje *Apple Macintosh*. Še v istem letu je *Microsoft* kupil *Forethought* in leta 1990 predstavil prvo različico za *Microsoft Windows 3.0* in jo vključil v svojo zbirko pisarniških orodij *Microsoft Office*. [19]

Šele najnovejša različica, *PowerPoint 2007*, je prinesla nekaj podpore za strojno pospešeno grafiko na osnovi grafične knjižnice *Direct3D*. Vseeno pa je nabor posebnih 3-D učinkov v osnovni namestitvi skromnejši od tistega v prej omenjenem *Apple Keynote*. *PowerPoint* deluje tako v operacijskem sistemu *Windows* kot tudi *Mac OS X*. V operacijskem sistemu *Linux* pa ga lahko poganjamo le s pomočjo posnemovalnikov, vendar se to praviloma negativno odraža na hitrosti in zanesljivosti delovanja. *PowerPoint* enako kot *Apple Keynote* omogoča tako pripravo predstavitve kot tudi njeno prikazovanje, je zaprtokoden in še nekoliko dražji. Ima svoj lasten datotečni zapis in v osnovni namestitvi ne podpira izvoza v *PDF* (poglavje 1.3).

Crystal Graphics PowerPlugs za Microsoft PowerPoint

Crystal Graphics PowerPlugs je zbirka dodatkov za *Microsoft PowerPoint*, ki uporabniku pri pripravi predstavitev omogoči uporabo množice novih gradnikov ter učinkov. Med drugim omogoča tudi učinek obračanja kocke za prehajanje med predstavitvenimi stranmi po zgledu prej omenjenega programa *Apple Keynote*.

OpenOffice.org Impress

OpenOffice.org Impress je del odprtokodne zbirke pisarniških orodij *OpenOffice.org*. Ta je nastala iz zbirke *StarOffice*, ki jo je razvilo podjetje *StarDivision* in leta 1999 prevzel *Sun Microsystems*. Deluje v večini današnjih operacijskih sistemov, kot so *Windows*, *Linux*, *Solaris*, *BSD*, *Mac OS X* ...

Enako kot *PowerPoint* in *Keynote* omogoča tako pripravo kot tudi prikaz predstavitev. Podpira večino sorodnih datotečnih zapisov, v osnovi pa uporablja *OASIS OpenDocument* zapis (poglavje 1.3). Pomanjkljivost je, da ne izkorišča strojno pospešene grafike, zato je dinamika predstavitev pogosto precej skromna.

Google Docs Presentation

Google Docs Presentation je zanimiv predvsem, ker je v celoti narejen kot spletna aplikacija in od uporabnika zahteva le sodoben spletni brskalnik. Omogoča tako urejanje kot tudi prikazovanje predstavitev, podpira različne datotečne zapise in je za uporabo brezplačen. Zaenkrat še ne omogoča nobene dinamike.

Adobe Acrobat Reader

Adobe Acrobat Reader je v osnovi prkazovalnik poljubnih *PDF* dokumentov, toda služi lahko tudi prikazovanju predstavitev. Ne omogoča dinamike, je pa lahko toliko bolj interaktiven. Od različice 7.0 podpira prikaz 3-D vsebine, ki jo uporabnik lahko s pomočjo strojno podprte grafike med drugim vrti in prikazuje iz različnih zornih kotov.

PDF Cube

PDF Cube je zelo enostaven prikazovalnik *PDF* dokumentov, ki prehaja med stranmi z učinkom obračanja kocke po zgledu programa *Apple Keynote*. Program je osnovan na grafični knjižnici *OpenGL*, za upodabljanje (angl. *rendering*) strani *PDF* dokumenta pa uporablja knjižnico *Poppler* (poglavje 1.4.4). V ukazni vrstici mu kot parametre določimo strani, ki naj za svoj prikaz uporabijo omenjeni učinek. Poleg pomikanja na naslednjo in prejšnjo stran *PDF Cube* omogoča še povečavo petih odsekov strani z učinkom zveznega približevanja in oddaljevanja.

Deklarativno animirane in interaktivne predstavitve

Grafični uporabniški vmesnik sicer uporabniku precej poenostavi delo s predstavitvami, vendar mu tudi precej omeji možnosti za njihovo pripravo. Za doseganje večje interaktivnosti in zahtevnejše dinamike je deklarativen pristop še vedno neizbežen. Nekatera orodja omogočajo pripravo tako imenovanih makrojev. Gre za programe, ki jih lahko sprožijo razni dogodki, povezani s prikazovanjem predstavitve in predstavitvi dodajajo večji pridih interaktivnosti.

Priprava teh makro programov je precej nerodna in neučinkovita, posebej ko želimo doseči poljubno kompleksno, po možnosti celo 3-D dinamiko. Kot alternativo je Douglas Zongker leta 2003 na simpoziju o računalniški grafiki, *ACM SIGGRAPH*, predstavil sistem *Slithy* [23]. Gre za skupek *Python* knjižnic, ki za upodabljanje strojno pospešene 3-D grafike uporablja knjižnico *OpenGL*, za upodabljanje besedil pa knjižnico *FreeType*. Uporabnik predstavitev tako v celoti spiše v *Pythonu*.

Za konec naj omenim še možnost uporabe *Shockwave Flash Object (SWF)* datotek, ki jih pripravimo s programskimi orodji, kot sta na primer *Flash* in *Ming*. Za prikaz takih predstavitev potrebujemo poseben predvajalnik, ki trenutno še ne podpira strojno pospešene 3-D grafike.

1.3 Datotečni zapisi

Datotečnih zapisov (angl. *file format*) za predstavitvene dokumente je precej, zato bom tu naštel le nekaj najpogostejših. Zaenkrat najbolj razširjen zapis je *Microsoft PowerPoint Template (.ppt)* oziroma *Microsoft PowerPoint Slideshow (.pps)*. Gre za zaprt binaren zapis, uporabljen v starejših različicah aplikacije *Microsoft PowerPoint* in naj bi ga v prihodnosti nadomestil *Office Open XML (.pptx)*.

Office Open XML temelji na odprtem standardu *Ecma-376* s konca leta 2006, ki ga je *Microsoft* razvil kot protiutež zapisu *OASIS OpenDocument (.odp)*. V različici 2007 aplikacije *Microsoft PowerPoint* je *Office Open XML* privzet datotečni zapis, za podporo v starejših različicah pa že obstajajo razširitve.

OASIS OpenDocument je odprt standard od leta 2005. Le kak teden pred sprejetjem standarda *Ecma-376* pa je s koncem leta 2006 formalno postal potrjen tudi kot *ISO* standard *ISO/IEC 26300:2006*. Za njim stojita predvsem *Sun Microsystems* in *IBM*, podpira pa ga vrsta podjetij, med drugim *Google*, *Novell*, *Oracle* in *Red Hat*, ki se združujejo v tako imenovan *OpenDocument Format Alliance*. Tako *Office Open XML* kot tudi *OASIS OpenDocument* sta stisnjena *XML* zapisa.

V aplikaciji *OpenOffice.org Impress* je od različice 2.0.0 dalje privzet datotečni zapis *OASIS OpenDocument*, ki je izpodrinil dotedanji *OpenOffice.org Presentation (.sxi)*. *Apple Keynote* zaenkrat še vedno prisega na lasten zapis *Keynote Presentation (.key)*, a podpira tudi druge oblike.

Adobe Portable Document Format (PDF)

Adobe Portable Document Format (.pdf) je edina oblika datotečnega zapisa, ki sem jo podprl v aplikaciji, razviti v sklopu tega diplomskega dela. Gre za odprt standard, namenjen predvsem pregledovanju oziroma tiskanju in manj popravljanju obstoječih dokumentov. Uporablja se za najrazličnejše vrste dokumentov, med drugim tudi za multimedijske predstavitve. Razvil ga je *Adobe* leta 1993 in je sčasoma postal *de facto standard* za dokumente, namenjene tiskanju na medmrežju. Najnovejša različica, *PDF* 1.7, naj bi postala tudi *ISO* standard, ki ima trenutno v fazi osnutka ime *ISO 32000*.

PDF dokument lahko vsebuje tekst, nabore znakov (angl. *font*), rasterske in vektorske slike, notranje in zunanje povezave, kazalo ter celo interaktivne 3-D gradnike, ki se jih da vrteti in ogledovati iz različnih zornih kotov.

1.4 Izbor tehnologij

Sedaj bom na kratko predstavil tehnologije, med katerimi sem se odločal v fazi arhitekture. Izbiro nekoliko omejuje zahteva, da aplikacija deluje vsaj na operacijskih sistemih *Microsoft Windows* in *Linux*. Druga omejitev je licenciranje izbranih tehnologij, ki morajo biti do te mere združljive, da bo lahko končna rešitev objavljena pod katero izmed odprtokodnih licenc (*GPL*, *EPL*, *BSD* ...).

1.4.1 Razvojno okolje

Prenosljivosti med operacijskimi sistemi sem najpreprosteje ugodil z izbiro programskega jezika *Java*. Za integrirano razvojno okolje (angl. *integrated development environment – IDE*) pa sem vzel *Eclipse IDE*, ki sem ga bil vajen že od prej.

1.4.2 Programsko ogrodje

Pri pripravi uporabniškega vmesnika sem imel na razpolago orodjarni gradnikov (angl. *widget toolkit*) *AWT* in *Swing*, ki sta del standardnega grafičnega ogrodja okolja *Java* (*JFC*), ali pa *SWT*, ki je sestavni del *Eclipse RCP*. Predvsem iz estetskih razlogov sem izbral slednjo, saj temelji na grafičnem vmesniku operacijskega sistema. Preko vmesnika *JNI* namreč uporablja knjižnice *GTK*, *Win32*, *Carbon* oziroma *Motif* in zato daje podoben izgled kot aplikacije, razvite za domače (angl. *native*) okolje.

Razmerje med *Eclipse IDE*, platformo *Eclipse* in *Eclipse RCP* bo bolje razloženo v poglavju 2.2. Poleg knjižnice *SWT Eclipse RCP* prinese še druge koristne komponente za razvoj odjemalskih aplikacij, kot je *JFace*, ki omogoča uporabo *SWT* gradnikov z model-pogled-krmilnik (angl. *model-view-controller – MVC*) načrtovalskim vzorcem (angl. *design pattern*), ali pa denimo komponenta, ki skrbi za življenjski cikel aplikacije in vpelje koncept vtičnikov (angl. *plugin*), osnovan na *OSGi specifikaciji* (poglavje 2.1).

1.4.3 Knjižnice za delo s 3-D grafiko

Knjižnice za delo s 3-D grafiko bom razdelil v dve skupini glede na njihov programski vmesnik (angl. *application programming interface – API*). Knjižnice z nizkonivojskimi programskimi vmesniki ukaze za delo s 3-D grafiko sproti posredujejo do gonilnika grafične kartice. Visokonivojski programski vmesniki pa omogočajo pripravo objektnega opisa scene, ki se šele kasneje na zahtevo izriše z uporabo ukazov nizkonivojskih knjižnic.

Nizkonivojski programski vmesniki

Glede na nekatere arhitekturne omejitve sem za delo s 3-D grafiko uporabil nizkonivojski programski vmesnik *OpenGL*. Ta vmesnik je del odprtega standarda *OpenGL* [2], namenjenega razvoju grafične programske opreme. Predstavil ga je *Silicon Graphics Inc. (SGI)* leta 1992. Za nadzor nad standrdom je bil še istega leta ustanovljen *OpenGL Architecture Review Board (ARB)*, ki so ga sestavljala zainteresirana podjetja. Leta 2006 je bil nadzor predan konzorciju *Khronos Group*. *OpenGL* je najbolj razširjen pri aplikacijah za računalniško podprto načrtovanje (angl. *computer aided design – CAD*), navidezni resničnosti, raznih vizualizacijskih aplikacijah ter simulatorjih. Predvsem pri video igricah pa ima močnega tekmeca v programskem vmesniku *Direct3D*, ki ga je za okolje *Windows* razvil *Microsoft*. *OpenGL* je podprt na vseh večjih operacijskih sistemih, med drugim *Microsoft Windows*, *Mac OS X*, *UNIX*, *OS/2* in *Linux*. Ima povezave do programskih jezikov *C*, *C++*, *Fortran*, *Python*, *Ada*, *Perl*, *Java* ...

Direct3D je del širšega programskega vmesnika, *DirectX*, ki poleg upodabljanja 3-D grafike skrbi še za ostala opravila, povezana z multimedijo (2-D grafika, video, zvok, podpora za igralne palice ...) [1]. V primerjavi z vmesnikom *OpenGL* je njegova glavna pomanjkljivost zaprtost standarda, kar otežuje razvoj knjižnic in gonilnikov za druge operacijske sisteme. Podprt je na operacijskem sistemu *Microsoft Windows* in igralni konzoli *XBox*.

Omenjena standarda s svojima programskima vmesnikoma omogočata abstrakcijo strojne opreme in razvijalcem rešujeta težave, povezane z raznolikostjo podpore za strojno pospešeno 3-D grafiko. Poleg tega *OpenGL* nudi še programsko posnemanje (angl. *software emulation*) tistih funkcionalnosti, ki jih strojna oprema ne podpira.

Knjižnice, izvedene po omenjenih standardih, praviloma služijo tudi kot zasnova za visokonivojske programske vmesnike za delo s 3-D grafiko.

Visokonivojski programski vmesniki

Programski vmesnik *OpenGL* zahteva zelo proceduralen pristop za programiranje 3-D grafike. *Direct3D* je sicer nekoč podpiral tako imenovan zadržan način (angl. *retained mode*), ki je omogočal visokonivojsko programiranje, toda ta način je bil kasneje opuščen. Danes deluje le še v takojšnem načinu (angl. *immediate mode*), ki podobno kot *OpenGL* nudi zgolj neposreden vmesnik do funkcij grafičnga vmesnika. Nekoliko si sicer lahko pomagamo tudi s knjižnicami, kot sta *GLU* za *OpenGL* in *D3DX* za *Direct3D*, ki poenostavijo nekatere bolj zapletene operacije.

Pogosto pa za upodabljanje 3-D grafike uporabimo katero izmed knjižic, ki z visokonivojskimi programskimi vmesniki podpirajo tako imenovane grafe scene (angl. *scene graph*) [20]. Graf scene je strutkura, ki logično ali pa včasih tudi prostorsko razčlenjuje predstavitev scene. Gre za acikličen graf, večinoma drevo. Operacije, izvedene na posameznih vozliščih, se praviloma odrazijo na celotnem poddrevesu. Denimo geometrična transformacija vozila se smiselno prenese tudi na motor in kolesa, ki v grafu scene predstavljajo potomce vozlišča vozila. Te pa lahko nato posebej transformiramo glede na njihov položaj na vozilu.

Programski vmesniki za delo z grafi scen so pogosto del igralnih pogonov (angl. *game engine*), ki poleg upodabljanja 3-D grafike omogočajo še detekcijo trkov, delo z zvokom, umetno inteligenco, animacijo ... Primeri tovrstnih igralnih pogonov so *RealmForge*, *Ogre*, *jMonkey Engine* ...

OpenGL v programskem jeziku *Java*

Povezave za nizkonivojski programski vmesnik *OpenGL* v jeziku *Java* omogočata knjižnici *JOGL* in *LWLGL*. Gre za odprtokodni knjižnici, ki preko vmesnika *JNI* zgolj delegirata ukaze do knjižnice *OpenGL* v domačem okolju. Na razpolago so tudi visokonivojske knjižnice, ki temeljijo na grafih scen, kot so *Java3D*, *Aviatrix3D*, *jMonkey Engine* ... Za izdelavo te diplomske naloge sem izbral *JOGL*, saj sem si želel bolje spoznati programski vmesnik *OpenGL*.

1.4.4 Knjižnice za upodabljanje strani *PDF* datotek

V tej diplomski nalogi sem se posvetil predvsem prikazovanju *PDF* dokumentov, saj je za upodabljanje teh na razpolago precej različnih knjižnic. Prvo rešitev sem izvedel s knjižnico *jPedal*, ki je izdana pod licenco *GPL* in je v celoti narejena v programskem jeziku *Java*. Ta rešitev deluje tako na operacijskem sistemu *Linux* kot tudi *Microsoft Windows*. Njena težava je počasnost, saj posamezne strani lahko upodablja tudi nekaj sekund, kar zelo moteče vpliva na delovanje celotne aplikacije. Deloma ta problem rešujem s predpomnilnikom in upodabljanjem strani vnaprej, a ne v zadovoljivi meri. Druga težava pa je slaba kvaliteta upodobljenih strani, ki pogosto opazno odstopajo od rezultatov ostalih, bolj razširjenih prikazovalnikov. Poleg knjižnice *jPedal* velja omeniti še *PDFBox*, izdan pod licenco *BSD*, ter zaprtokodno knjižnico *ICEPdf*, ki sta obe prav tako narejeni za okolje *Java*. Teh knjižnic nisem preizkusil.

Zaradi opisanih težav sem pripravil še različico, ki upodablja strani z uporabo knjižnice *Poppler*. *Poppler* je prav tako izdan pod licenco *GPL*, napisan je v programskem jeziku *C++* in se je razvil iz knjižnice *Xpdf*. Pri tej izvedbi sem moral pripraviti vmesnik *JNI*, ki iz programskega jezika *Java* omogoča klice v domače okolje. Rezultati so bistveno boljši, več o sami izvedbi pa bo govora v poglavju 2.5.2. Alternativno bi lahko uporabil tudi nekoliko manj znano *GPL* knjižnico *Fitz/MuPDF*.

1.5 Zgradba dela

V pričujočem delu bo najprej govora o izvedbi same aplikacije za prikaz predstavitvev, čemur je posvečeno drugo poglavje. Prvi del poglavja predstavi arhitekturo aplikacije in koncepte, ki jih vpeljejo nekatere arhitekturne odločitve, kot je na primer izbira platforme *Eclipse RCP*. V nadaljevanju je govora o nizkonivojski zasnovi in izvedbi nekaterih konkretnih funkcionalnosti. Poglavje se zaključi z razlago integracije knjižnic za upodabljanje *PDF* dokumentov.

Tretje poglavje govori o načinu prikazovanja predstavitev z uporabo 3-D grafike. Najprej je poudarek na geometriji zastavljenega problema in z njim povezanih specifičnih lastnostih kocke oziroma bolj splošno kvadra. Nato je prikazano, kako se kocka upodablja z uporabo nizkonivojske knjižnice za delo s 3-D grafiko, *OpenGL*. Preostanek poglavja razloži še objektni model za predstavitev scene in animacije. V kontekstu animacije so pojasnjeni nekateri matematični prijemi, kot so uporabljene interpolacijske funkcije, preračunanje prostorskih stanj ter Bézierove krivulje in B-zlepki za glajenje trajektorij gibanja. Na koncu poglavja je razložena podprta interakcija uporabnika s kocko.

V zaključku dela so povzeti rezultati, pomanjkljivosti izvedene rešitve ter možnosti za nadaljnje izboljšave. Dodatek A na kratko predstavi kvaternione ter nekatere operacije nad njimi.

Poglavje 2

Izvedba aplikacije

Cilj diplomskega dela je bila izdelava aplikacije za prikazovanje predstavitev, projiciranih na ploskve kocke, izrisane s pomočjo strojno pospešene 3-D grafike. Aplikaciji sem dal ime *jPresenter* in lahko deluje bodisi samostojno ali pa kot dodatek k obstoječim aplikacijam, ki temeljijo na *Eclipse RCP*. V skladu s *platformo Eclipse* (angl. *Eclipse Platform*) [5] je *jPresenter* sestavljen iz vtičnikov, ki bolj ali manj neodvisno prispevajo vsak svoj del funkcionalnosti. Zasnova aplikacije omogoča, da dodajanje podpore za novo vrsto datotečnega zapisa ali pa za drugačno izvedbo prikazovalnika zahteva le pripravo ustreznega vtičnika.

2.1 Vtičniki v *Eclipse* in *OSGi* specifikacija

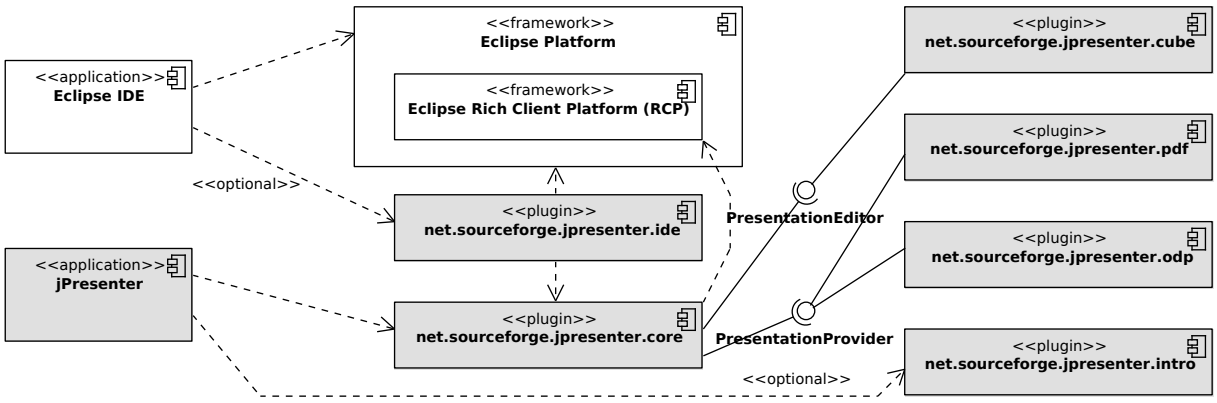
Eclipse IDE in ostale aplikacije, ki temeljijo na *Eclipse RCP*, so v osnovi sestavljene iz vtičnikov. Infrastruktura, ki skrbi za življenjski cikel posameznih vtičnikov in njihovo medsebojno interakcijo, je izvedena v skladu z *OSGi* specifikacijo. Gre za specifikacijo dinamično modularnega sistema za programsko okolje *Java*.

OSGi specifikacija [18] ponuja standardiziran pristop za gradnjo aplikacij iz majhnih, ponovno uporabnih (angl. *reusable*) in sodelojočih komponent. Nudi dinamično sestavljanje in menjavo komponent brez potrebe po ponovnem zagonu aplikacije. Servisno orientirana arhitektura *OSGi* sistema omogoča avtomatsko medsebojno zaznavanje komponent za njihovo sodelovanje.

Osrednji del *OSGi* specifikacije je večplastno ogrodje, ki skrbi za izvajanje (angl. *execution*), modularnost komponent (angl. *modularity*), življenjski cikel komponent (angl. *life cycle*) in servisni register (angl. *service registry*). Modularnost je zagotavljena z lastnim nalagalnikom razredov (angl. *class loader*), ki na nivoju posameznih komponent omogoča zaščito notranjih, četudi javnih razredov pred nenamensko rabo iz drugih komponent. Življenjski cikel komponent predstavlja njihovo dinamično nameščanje, zaganjanje, ustavljanje, nadgrajevanje in odstranjevanje. Servisni register pa omogoča medsebojno lociranje in interakcijo med komponentami, upoštevajoč dinamičnost njihovega nameščanja.

Platforma Eclipse [10] temelji na *OSGi* specifikaciji predvsem glede zagotavljanja modularnosti in življenjskega cikla komponent oziroma vtičnikov. Servisno podporo za interakcijo med vtičniki pa nadomešča lasten sistem razširitvenih točk (angl. *extension point*). Nekateri vtičniki tako določajo razširitvene točke, na katere lahko drugi vtičniki prispevajo svoje razširitve (angl. *extension*). Primer razširitvene točke ima denimo vtičnik za nastavitveno okno, ki omogoča uporabniku prilagajanje programskih nastavitev. Drugi vtičniki pa lahko prispevajo svojo nastavitveno stran znotraj nastavitvenega okna kot razširitev omenjene razširitvene točke. Definicije razširitev in razširitvenih točk podamo v datoteki `plugin.xml`.

2.2 Razčlenitev na vtičnike



Slika 2.1: *UML* komponentni diagram za *jPresenter*.

Razčlenitev aplikacije *jPresenter* predstavlja komponentni diagram na sliki 2.1. Sive komponente so specifične za aplikacijo *jPresenter*, bele pa so del *platforme Eclipse*. Iz diagrama je razviden odnos med ogrodjem *platforme Eclipse* in *Eclipse RCP*. *Eclipse RCP* je skupek vsesplošno uporabnih vtičnikov, ki med drugim skrbijo za uporabniški vmesnik (*SWT* in *jFace*), *OSGi* integracijo in *Eclipse Runtime*. *Platforma Eclipse* pa poleg *Eclipse RCP* vsebuje še vtičnike, specifične za razvojna okolja, na primer za *Eclipse IDE*. V ta sklop sodijo vtičniki za razhroščevalnik, pomoč, iskanje, ekipno delo ... Aplikaciji *Eclipse IDE* in *jPresenter* sta fizično definirani v vtičnikih `org.eclipse.ide` oziroma `net.sourceforge.jpresenter.core`, na sliki pa sta prikazani kot posebni logični komponenti. Črtkane puščice kažejo medsebojno odvisnost oziroma uporabo komponent. Četudi so vse te povezave vzpostaljene preko standardnih razširitvenih točk, ki so del *platforme Eclipse*, sta razširitveni točki `PresentationEditor` in `PresentationProvider` na sliki posebej izpostavljeni. Več o tem v nadaljevanju.

2.2.1 Osnovni vtičnik

Osnovni vtičnik je poimenovan `net.sourceforge.jpresenter.core`. Vsebuje tako infrastrukturo aplikacije kot tudi uporabniški vmesnik.

Uporabniški vmesnik

V kontekstu uporabniškega vmesnika so v tem vtičniku definirani *aplikacija*, *perspektiva*, nekateri *pogledi* in osnovna izvedba *urejevalnika*. Izrazoslovje je povzeto iz *Eclipse RCP* in ga velja na kratko razložiti.

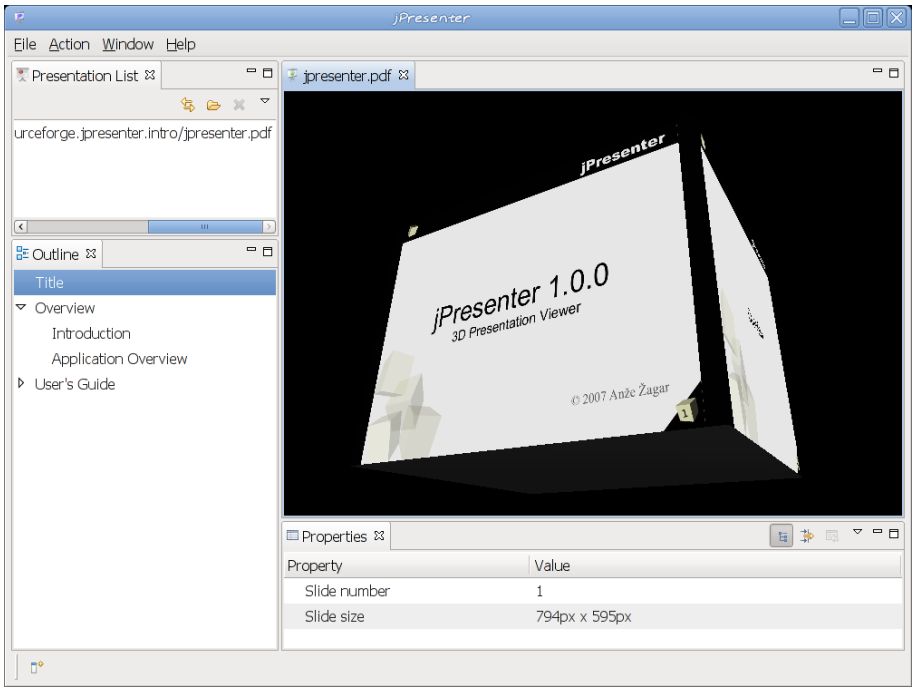
- **Urejevalnik** je odložljivo (angl. *dockable*) okno, praviloma vezano na posamezen, vnaprej določen primerek odprtega dokumenta. Odlaga se lahko le v štirikotno območje glavnega aplikacijskega okna, ki je rezervirano za urejevalnike. Odprti urejevalniki se bodisi prekrivajo ali pa si razdelijo omenjeno območje. Tisti, ki ostanejo prekriti, so dostopni s pomočjo zavihkov (angl. *tab*).
- **Pogledi** lahko poljubno zapolnjujejo preostali prostor aplikacijskega okna. Vsebina posameznega pogleda se praviloma prilagaja trenutno izbranemu urejevalniku oziroma pripadajočemu dokumentu.
- **Perspektiva** določa izhodiščno razporeditev pogledov in prostora za urejevalnike. Na razpolago imamo lahko več perspektiv, ki jih dinamično preklapljamo med delovanjem aplikacije, odvisno od trenutnega konteksta uporabe.
- **Aplikacija** je definirana s privzeto perspektivo ter nekaterimi atributi, vezanimi na njeno označevanje, kot so na primer ime, ikona, avtorstvo, licence, predstavitveno okno ...

Slika 2.2 prikazuje privzeto perspektivo aplikacije *jPresenter*. Levo je pogled *Presentation List*, ki drži seznam odprtih predstavitvenih dokumentov. Pod njim je *Outline*, ki prikazuje razčlembo trenutno izbranega dokumenta, na primer kazalo v primeru predstavitvenih dokumentov. Desno spodaj je pogled *Properties*, ki za zadnji izbran objekt v katerem koli drugem oknu (npr. dokument, stran, poglavje ...) izpiše nekatere njegove lastnosti. Slednja dva pogleda sta splošna in sta že del *Eclipse RCP*. Zagotoviti pa jima je bilo potrebno podporo za predstavitvene dokumente, o čemer bom nekoliko več spregovoril v poglavju 2.3.

Preostalo območje je namenjeno trenutno prikazanim urejevalnikom. Edini odprt urejevalnik na sliki prikazuje dokument z imenom `jpresenter.pdf`. V našem primeru gre sicer za prikazovalnik predstavitve in je beseda urejevalnik morda rahlo zavajajoča. Ta beseda je vzeta iz okolja *Eclipse*, kjer so bili prvotno urejevalniki namenjeni predvsem urejanju izvorne kode in drugih besedilnih dokumentov.

Infrastruktura aplikacije

Podrobnejšo zasnovo bom obdelal v poglavju 2.4. Zaenkrat pa bom dal več poudarka na interakciji z ostalimi vtičniki. Ob odprtju dokumenta mora aplikacija najprej ugotoviti,



Slika 2.2: Posnetek aplikacije *jPresenter*.

za katero vrsto datotečnega zapisa gre, in uporabiti ustrezen vtičnik. Za prvi del poskrbi že *Eclipse RCP*. Ta preišče seznam podprtih datotečnih zapisov, ki jih posamezni vtičniki prijavijo preko razširitvene točke `org.eclipse.core.contenttype.contentTypes`. Vrne tistega, ki se sam prepozna za najustreznejšega, bodisi na osnovi končnice ali pa vsebine datoteke. *Eclipse RCP* nato poišče še ustrezen urejevalnik, ki je prijavljen najvišje na prioritetnem seznamu za urejanje ugotovljenga datotečnega zapisa. Vsi datotečni zapisi za predstavitve morajo zato dedovati iz abstraktnega datotečnega zapisa `net.sourceforge.jpresenter.content-type.presentation`, ki je določen v našem osnovnem vtičniku in mora biti prijavljen pri vseh izvedbah urejevalnikov za prikaz predstavitev. *Eclipse RCP* nato inicializira urejevalnik in mu poda ročko na datoteko.

Urejevalniki predstavitev praviloma dedujejo iz abstraktne izvedbe urejevalnika `PresentationEditor`, ki je del osnovnega vtičnika. `PresentationEditor` že poskrbi, da se glede na vrsto datotečnega zapisa poišče pripadajoči ponudnik predstavitve. Išče se med prijavljenimi razširitvami razširitvene točke `net.sourceforge.jpresenter.presentationProviders`. Ponudnik predstavitve nato služi kot vmesnik za delo s predstavitvami, opisanimi s podatkovnimi strukturami, ki so prav tako del osnovnega vtičnika. Ponudnike predstavitev si bomo podrobneje ogledali v poglavju 2.4.3.

2.2.2 Vtičniki za podporo predstavitvenim datotečnim zapisom

Kot je mogoče razbrati že iz poglavja 2.2.1, mora vtičnik za podporo novega predstavitvenega datotečnega zapisa prijaviti dve razširitvi. Prva je datotečni zapis `org.eclipse.core.contenttype.contentTypes`, ki mora dedovati iz `net.sourceforge.jpresenter.content-type.presentation`. Druga pa je ponudnik predstavitev `net.sourceforge.jpresenter.presentationProviders`. Razširitvi za ponudnika predstavitev je potrebno določiti še izvedbo `IPresentationProvider` vmesnika, ki priskrbi vso logiko za tolmačenje danega datotečnega zapisa.

Vtičnika za podporo *PDF* predstavitev

V sklopu te diplomske naloge sta bila narejena dva vtičnika, ki priskrbita podporo za prikaz *PDF* dokumentov. Gre za vtičnika `net.sourceforge.jpresenter.pdf.jpedal` in `net.sourceforge.jpresenter.pdf.poppler`. Za upodabljanje posameznih strani dokumenta in razčlembo vsebine uporabljata knjižnici *jPedal* oziroma *Poppler*. Podrobneje bom o tem spregovoril v poglavju 2.5.

2.2.3 Vtičniki za prikaz predstavitev

Dokumente praviloma prikazujemo v urejevalnikih. Funkcija vtičnika za prikaz predstavitev je ravno v tem, da priskrbi izvedbo ustreznega urejevalnika za predstavitvene dokumente. To najlažje dosežemo z dedovanjem iz abstraktnega razreda `PresentationEditor`, ki je na razpolago v osnovnem vtičniku. Nov urejevalnik pa prijavimo kot razširitev razširitvene točke `org.eclipse.ui.editors`, ki ji določimo povezavo na podatkovni zapis `net.sourceforge.jpresenter.content-type.presentation`.

Vtičnik za prikaz predstavitev, projiciranih na ploskve kocke

Vtičnik za prikaz predstavitev, projiciranih na ploskve kocke, je ključni del te diplomske naloge. Poimenovan je `net.sourceforge.jpresenter.cube`, njegovemu delovanju pa je posvečeno celotno tretje poglavje.

2.2.4 Vtičnik za integracijo z *Eclipse IDE*

Omenjeno podporo za prikaz predstavitev želimo integrirati še v *Eclipse IDE*. V osnovi zadošča le kopiranje novih vtičnikov v `plugins` mapo, toda to deluje le za odpiranje predstavitvenih dokumentov neposredno iz datotečnega sistema. *Eclipse IDE* namreč operira s tako imenovanimi viri (angl. *resource*), ki sestavljajo nek odprt projekt. Viri sicer praviloma res predstavljajo neko fizično datoteko na disku, vendar jim je potrebno zagotoviti neposredno povezavo z logiko za rokovanje s predstavitvenimi dokumenti. To povezavo zagotavlja vtičnik `net.sourceforge.jpresenter.ide`, ki je posledično odvisen od celotne *platforme Eclipse* in zato ne sme biti vključen v samostojno aplikacijo *jPresenter*.

2.2.5 Vtičnik za uvodno predstavitev

Vtičnik za uvodno predstavitev, `net.sourceforge.jpresenter.intro`, ob prvem zagonu aplikacje *jPresenter* odpre predstavitveni dokument, ki na kratko razloži njeno uporabo. Prav tako v meni za pomoč doda opcijo za naknadno odprtje te predstavitve. Za delovanje potrebuje tudi vtičnik za podporo *PDF* predstavitev.

2.3 Pretvorniki tipov

V tem poglavju bomo spoznali koncept oziroma načrtovalski vzorec (angl. *design pattern*), ki služi kot pomembno orodje pri razširjanju funkcionalnosti v *platformi Eclipse*. Gre za pretvarjanje tipov [6], ki lahko s pomočjo posebnega registra tovarn pretvornikov za poljuben primerek (angl. *instance*) objekta vrne njegov pretvornik (angl. *adapter*) v nek drug vmesnik. Kadar izvorni primerek že sam izvaja (angl. *implements*) želeni vmesnik, je pretvorba običajno le enostavna prevedba tipa (angl. *typecasting*) in se v bistvu lahko vrne kar isti primerek. Dodana vrednost tega pristopa pa je, kadar želimo pretvarjati primerke, katerih izvedba v osnovi ne nudi potrebne funkcionalnosti oziroma jo nudi v drugačni obliki. Edina zahteva je, da izvorni primerek izvaja vmesnik `IAdaptable` ali pa razširja razred `PlatformObject`, kar pa velja za večino vmesnikov in razredov v *platformi Elipse*.

Oglejmo si na primeru vtičnika za integracijo z *Eclipse IDE*. Kot sem omenil v poglavju 2.2.4, je potrebno vzpostaviti povezavo med viri v *Eclipse IDE* in logiko za rokovanje s predstavitvenimi dokumenti. Težava je namreč, da bi neposredna povezava zahtevala odvisnost aplikacije *jPresenter* od celotne *platforme Eclipse*. Povezavo zato vzpostavi poseben vtičnik, ki registrira tovarno pretvornikov kot razširitev razširitvene točke `org.eclipse.core.runtime.adapters`. Pri registraciji se določi izvorni vmesnik, ki je lahko vmesnik za datotečne vire `IFile`, izhodni vmesnik `IFileStorage`, ki ga potrebuje *jPresenter*, ter izvedba tovarne pretvornikov – razred, ki izvaja vmesnik `IAdapterFactory`. Ob odprtju urejevalnika za prikaz predstavitev *jPresenter* z uporabo klica `(IFileStore) input.getAdapter(IFileStore.class)` pretvori podan vhodni primerek (`input`) v primerek, ki izvaja `IFileStorage`. Kot je bilo rečeno v zaključku prejšnjega odstavka, mora `input` izvajati vmesnik `IAdaptable`.

Iz opisanega koncepta je mogoče razbrati, da posamezen izvorni primerek lahko izvaja tudi svojo lastno izvedbo metode `getAdapter(Class)`. Na tak način lahko pretvorbo določa tudi vsak tip po svoje, mimo registra tovarn pretvornikov, ali pa tovarno pretvornikov uporabi zgolj kot rezervno opcijo. Večina razredov v *platfromi Eclipse* deduje iz razreda `PlatformObject`, ki uporablja omenjeni register.

Ta pristop je uporabljen tudi pri pretvarjanju oblik upodobljenih strani predstavitev, o čemer bo govora v poglavju 2.4.2.

2.4 Zasnova aplikacije

Zaradi preobsežnosti zasnove celotne aplikacije se bom v tem poglavju osredotočil le na tiste dele, ki pomembneje vplivajo na funkcionalnost.

2.4.1 Podatkovne strukture za hranjenje predstavitev

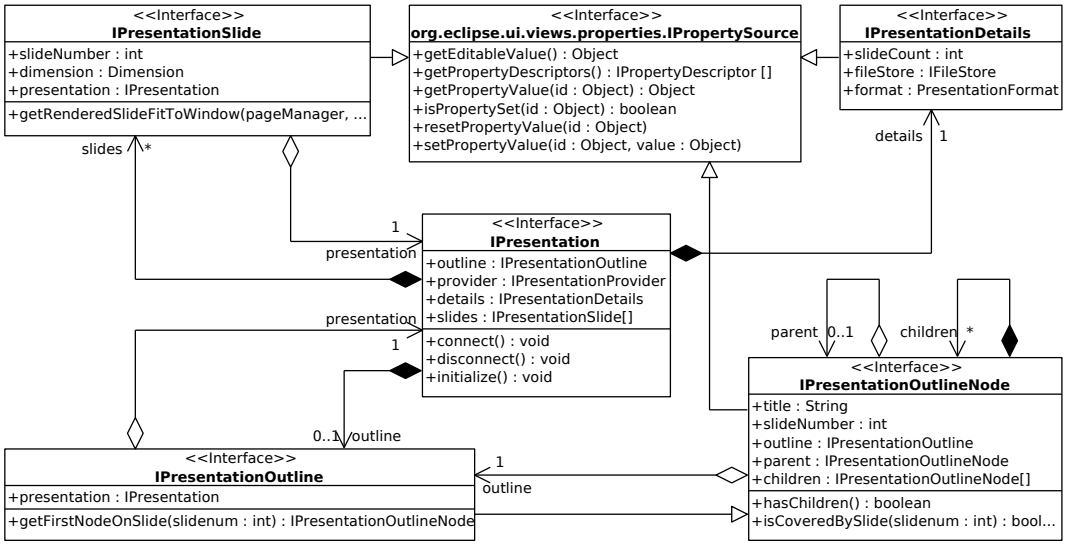
Podatkovne strukture za hranjenje prestavitev so del osnovnega vtičnika. Določene so s pomočjo vmesnikov, ki so javno dostopni iz vseh odvisnih vtičnikov. Razredi, ki izvajajo te vmesnike, so po večini skriti, njihovi primerki pa se generirajo s pomočjo tovarniškega načrtovalskega vzorca (angl. *factory design pattern*). Diagram razredov 2.3 prikazuje vmesnike, ki opisujejo primerke odprtih predstavitvenih dokumentov. Razredi, ki izvajajo te vmesnike, so prikazani na diagramu 2.4. Slednji diagram prikazuje še vmesnik za ponudnike predstavitev, `IPresentationProvider`.

Inicializacija podatkovnih struktur se vrši na zahtevo (angl. *lazy initalization*), kar pomeni, da se podatki o predstavitvenem dokumentu in posameznih straneh dokumenta ugotavljajo šele, ko se dejansko potrebujejo. Ugotavlja jih ponudnik predstavitev, ki je specifičen za posamezno vrsto datotečnega zapisa (poglavje 2.2.2). Alternativno lahko celotno inicializacijo izvedemo takoj s klicem metode `initialize()` na primerku predstavitve.

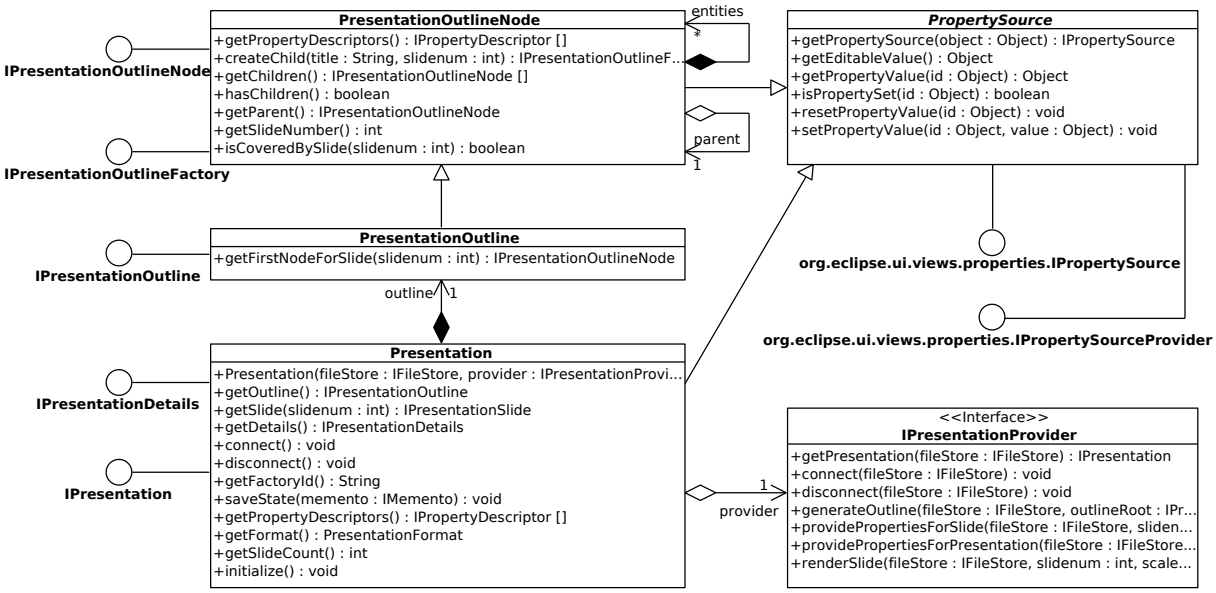
Predstavitev je predstavljena z vmesnikom `IPresentation`. Ta vmesnik nudi dostop do posameznih strani dokumenta, kazala ter podrobnosti o dokumentu. Vsaka stran dokumenta (`IPresentationSlide`), podrobnosti o dokumentu (`IPresentationDetails`) ter posamezna poglavja v kazalu (`IPresentationOutlineNode`) preko vmesnika `IPropertySource` ponujajo nek nabor lastnosti. `IPropertySource` je del *Eclipse RCP* in ga v kombinaciji z vmesnikom `IPropertySourceProvider` uporablja pogled *Properties* (poglavje 2.2.1) za prijavo lastnosti, ki naj jih ta pogled prikazuje. Lastnosti, ki jih hranimo na tak način, so denimo številka in velikost posamezne strani, vrsta datotečnega zapisa, število strani v dokumentu ter na primer naslov in številka začetne strani izbranega poglavja.

Vmesnik `IPresentation` razširja `IPersistableElement`, ki na diagramu 2.3 sicer ni prikazan. `IPersistableElement` je prav tako del *Eclipse RCP* in služi za shranjevanje stanja primerkov izvajajočega razreda. *Eclipse RCP* ga med drugim uporabi za shranjevanje seznama odprtih dokumentov (na primer za pogled *Presentation List*) in odprtih urejevalnikov ob zaprtju aplikacije. Ko se aplikacija naslednjič zažene, se vzpostavi prejšnje stanje, za kar je potrebno prijaviti še ustrezne razširitve razširitveni točki `org.eclipse.ui.elementFactories`. Te razširitve podajo izvedbo vmesnika `IElementFactory`, ki iz shranjenih podatkov nazaj naredi identične primerke prej odprtih dokumentov in urejevalnikov.

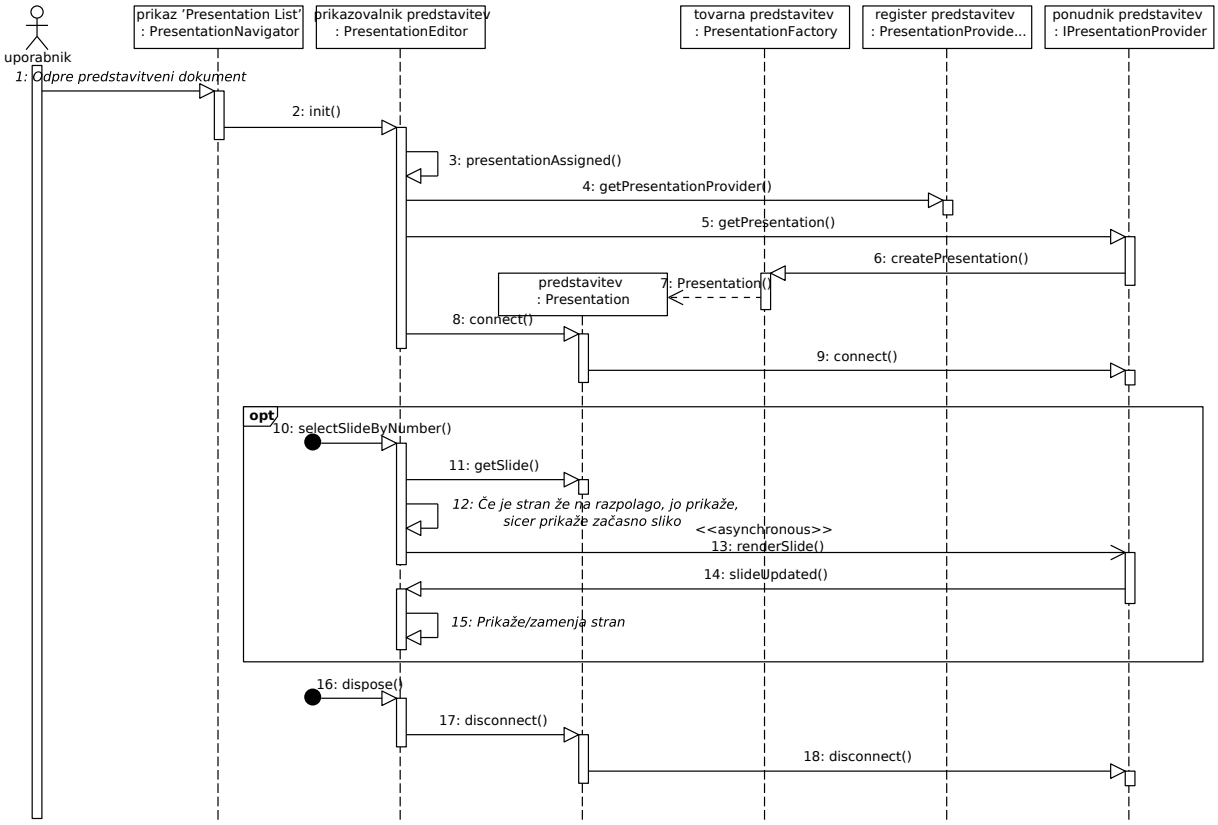
Primer delovanja in interakcije opisanih struktur podrobneje opisuje diagram 2.5. Gre za sekvenco dogodkov, ki se izvršijo ob odprtju predstavitvenega dokumenta in pripadajočega prikazovalnika (dogodki 1–9), ob morebitni zamenjavi trenutne strani (do-



Slika 2.3: *UML* diagram razredov podatkovnih struktur.



Slika 2.4: *UML* diagram razredov, ki izvajajo vmesnike iz diagrama 2.3.



Slika 2.5: *UML* diagram zaporedja pri življenjskem ciklu prikazovalnika predstavitev.

godki 10–15) ter ob zaprtju prikazovalnika (dogodki 16–18). Poudariti velja pomen metod `connect()` ter `disconnect()`, ki ponudniku predstavitev omogočata štetje aktivnih referenc na dan predstavitveni dokument. Ko število aktivnih referenc pade na nič, lahko ponudnik predstavitev sprosti pripadajoče vire. Pri *PDF* dokumentih je tak vir lahko razčlenjevalnik, ki ob prvem klicu `connect()` razčleni dokument in ob zadnjem `disconnect()` sprosti celotno razčlembo.

V sekvenci dogodkov 10–15 je prikazana asinhrona menjava strani. Torej, ko se zahteva prikaz nove strani, se najprej preveri, če ta v zadovoljivi velikosti že obstoji v predpomnilniku (angl. *cache*) (dogodek 12). Dogajanje, povezano s predpomnilnikom, bo podrobneje opisano v poglavju 2.4.2. V primeru, da ustrezne upodobitve strani še ni, se ponudniku predstavitev pošlje asinhrona zahteva za njeno upodobitev pri neki minimalni velikosti (dogodek 13). Začasno se prikaže bodisi upodobitev strani v manjši velikosti, če je bila taka najdena v predpomnilniku, ali pa posebna začasna stran (dogodek 12). Ko ponudnik predstavitve konča upodabljanje, prikazovalniku posredno vrne klic `slideUpdated()`, ki zamenja stran v njeno končno obliko (dogodka 14 in 15).

2.4.2 Upodobljene strani predstavitev

Slike strani se upodabljajo v vtičnikih za podporo predstavitvenim datotečnim zapisom (poglavji 2.2.2 in 2.4.3). Oblika predstavitve slike je lahko različna. Denimo knjižnica *jPedal* vrne upodobljeno stran kot primerek razreda *BufferedImage* iz knjižnice *AWT*. Podobno jo knjižnica *Poppler* preko vmesnika *JNI* vrne v obliki polja bajtov. Če bi hoteli stran prikazati v enostavnem pogledu ali pa urejevalniku v aplikaciji, ki kot rečeno sloni na *Eclipse RCP*, bi jo želeli pretvoriti v obliko primerka razreda *Image* iz knjižnice *SWT*. In nenazadnje, pri delu z *OpenGL* knjižnico, kjer moramo sliko kot teksturo naložiti v videopomnilnik, lahko to sliko pomnimo zgolj kot nek sklic na to teksturo. Slika mora torej biti zmožna menjati oblike, predpomnilnik pa naj jo v danem trenutku, če je to mogoče, hrani zgolj v najkoristnejši obiki.

Prav tako pomembno je vprašanje velikosti slike, ki se poda kot parameter metode `getRenderedSlideFitToWindow()` na vmesniku `IPresentationSlide`. S hranjenjem rasterskih slik visoke ločljivosti lahko hitro zapolnimo razpoložljiv pomnilnik, zato z velikostjo slike ne velja pretiravati. Glede na to, da gre za predstavitvene dokumente, ki se jih praviloma gleda iz daljše razdalje, si po eni strani lahko privoščimo nekoliko nižjo ločljivost. Po drugi strani pa urejevalnik za prikaz predstavitve lahko omogoča povečavo dela strani in prenizka ločljivost postane moteča. Izbira minimalne velikosti slike je tako prepuščena posamezni izvedbi urejevalnika, običajno pa je vezana na velikost predstavitvenega okna ali pa zaslonsko ločljivost.

Pretvarjanje oblik

Pretvarjanje oblik predstavitev slik sem omenil že na koncu poglavja 2.3, kjer sem opisal splošen koncept pretvornikov tipov v *Eclipse RCP*. Vse oblike slik so v bistvu razširitve abstraktnega razreda `ImageStorage`, ki izvaja vmesnik `IAdaptable` z uporabo registra tovarn pretvornikov. V celotni aplikaciji so trenutno izvedene naslednje štiri različice oblik predstavitev slik:

(i) `AWTImageStorage` je izveden v osnovnem vtičniku in služi kot ovojnica (angl. *wrapper*) za primerek razreda `BufferedImage` iz knjižnice *AWT*. Metoda `getAdapter()` neposredno izvaja pretvorbo v `SWTImageStorage` in `ByteArrayImageStorage` brez pomoči registra tovarn pretvornikov. Tako obliko predstavitve slike uporablja vtičnik za upodabljanje strani *PDF* dokumentov s knjižnico *jPedal*.

(ii) `SWTImageStorage` je prav tako del osnovnega vtičnika in ovija primerek razreda `Image` oziroma `ImageData` iz knjižnice *SWT*. Neposredno se zna pretvoriti v obliki `AWTImageStorage` in `ByteArrayImageStorage`. Ta oblika je smiselna za prikaz strani v *SWT* komponentah uporabniškega vmesnika. Uporablja jo preprosta izvedba urejevalnika `SimplePresentationEditor`.

(iii) `ByteArrayImageStorage` je ovojnica za navadno polje bajtov, ki predstavljajo sekvenco pikslov. Poleg polja bajtov drži še informacijo o širini in višini slike ter obliki

zapisa posameznega piksla. Prav tako je že del osnovnega vtičnika in se zna neposredno pretvoriti v `AWTImageStorage` ter `SWTImageStorage`. Uporablja ga vtičnik za upodabljanje strani *PDF* dokumenta s knjižnico *Poppler*, ki preko vmesnika *JNI* dobi upodobljeno stran v obliki polja bajtov.

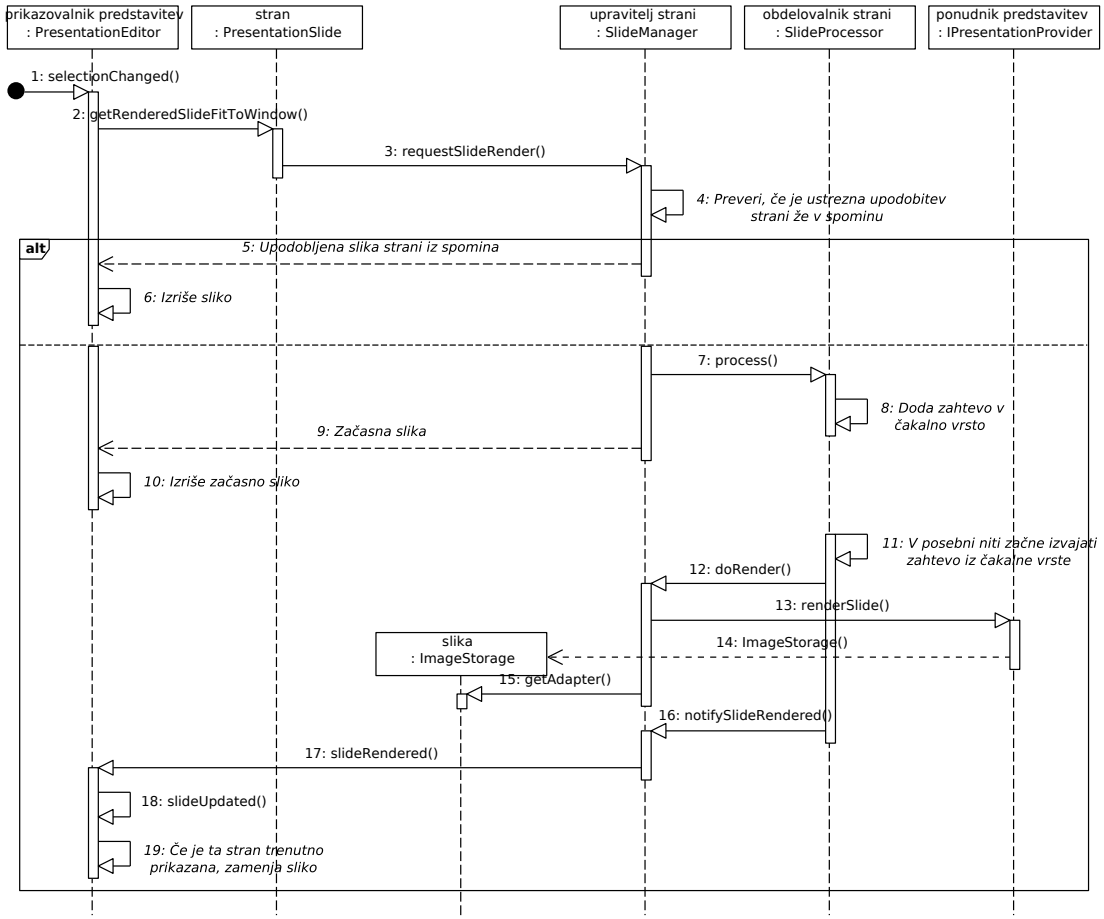
(iv) `GLImageStorage` je del vtičnika za prikaz predstavitev, projiciranih na ploskve kocke, o katerem bom več govoril v poglavju 3. Vsebuje zgolj indeks teksture v videopomnilniku, kjer je hranjena dejanska slika. Gre za končno obliko predstavitve slike in ne omogoča pretvorbe v druge oblike. Za pretvorbo iz ostalih oblik pa si mora registrirati dodatne tovarne pretvornikov, saj ostale oblike same po sebi te oblike ne poznajo.

Predpomnilnik

Samo upodabljanje strani je lahko časovno precej zahtevno. Konkretno *jPedal* knjižnjica lahko eno stran upodablja tudi več sekund. *Poppler* je sicer bistveno hitrejši, vendar vseeno lahko zmoti animacijo kocke, četudi ima nit, ki izvaja upodabljanje, najnižjo možno prioriteto. Vsaj deloma te probleme rešuje predpomnilnik upodobljenih strani. Vseeno pa ima predpomnilnik pri predstavitvenih dokumentih precej majhno dodano vrednost, saj predstavitvene strani praviloma obračamo od začetka proti koncu in se redko vračamo nazaj. Njegov učinek lahko nekoliko izboljšamo tako, da zahtevamo upodobitev naslednjih nekaj strani vnaprej. Idealno je, če se to stori takrat, kadar animiran prikazovalnik miruje.

Predpomnilnik je vezan na posamezen prikazovalnik predstavitev in je že vključen v abstraktnem razredu `PresentationEditor`. Alternativno bi lahko imeli en sam predpomnilnik za vse prikazovalnike hkrati. Prednosti takega pristopa bi bila bolje dorečena skupna omejitev razpoložljivih predpomnilniških kapacitet in možnost uporabe ene same upodobitve dane strani v več prikazovalnikih, ki prikazujejo isti predstavitveni dokument. Po drugi strani pa vsak prikazovalnik lahko zahteva drugačno obliko predstavitve slike in drugačno ločljivost, kar precej oteži izvedbo takega enotnega predpomnilnika.

Na diagramu zaporedja 2.6 je opisan postopek prikazovanja in upodabljanja neke strani v prikazovalniku predstavitev. Ta diagram v bistvu le podrobneje opisuje korake 10–15 iz diagrama 2.5. Vso logiko, povezano s predpomnilnikom, izvaja upravitelj strani. Kadar upodobitve strani, ki jo zahteva njegov prikazovalnik, ne najde v predpomnilniku (dogodek 4), delegira zahtevo na obdelovalnik strani in jo shrani v njegovo čakalno vrsto (dogodka 7 in 8). Nato prikazovalniku vrne začasno sliko, ki jo ta lahko prikazuje, dokler ustrezna upodobitev strani še ni na razpolago (dogodka 9 in 10). Obdelovalnik strani je samo eden za celotno aplikacijo in teče v svoji niti, ki zaporedoma izvaja zahteve iz čakalne vrste (dogodek 11). Zahtevo zanj izpolni ponudnik predstavitve, ki upodobljeno stran vrne kot nov primerek slike (dogodka 13 in 14). Vrnjeno sliko nato upravitelj strani pretvori v obliko, primerno za prikazovalnik, (dogodek 15) ter mu jo ponudi za izris (dogodki 17–19).



Slika 2.6: *UML* diagram zaporedja pri prikazu in upodobitvi predstavitvene strani.

2.4.3 Ponudniki predstavitev

Večkrat sem že omenil vmesnik `IPresentatioProvider`, ki je prikazan na diagramu 2.4. Izvedba tega vmesnika od knjižnice za razčlenjevanje predstavitvenih dokumentov pričakuje izgradnjo drevesne strukture za opis kazala in ugotavljanje nekaterih lastnosti predstavitvenega dokumenta oziroma posameznih strani. Večinoma so ti podatki opcijski, nujna sta le podatka o številu in velikosti strani. Knjižnica za upodabljanje pa mora biti zmožna pripraviti rastersko upodobitev posameznih strani v poljubni velikosti oziroma ločljivosti.

Kazalo dokumenta

Klic metode `generateOutline()` na ponudniku predstavitev pripravi kazalo v obliki drevesne strukture iz vozlišč tipa `PresentationOutlineNode`. Ta razred vozlišč izvaja vmesnika `IPresentationOutlineNode` ter `IPresentationOutlineNodeFactory`, kar je razvidno tudi iz diagramov 2.3 in 2.4. Prvi je namenjen pregledovanju drevesa, drugi pa služi

ponudniku predstavitev, da preko metode `createChild()` rekurzivno sestavi kazalo. Metoda `generateOutline()` pri pripravi kazala izhaja iz korenskega vozlišča, ki ga dobi kot vhodni parameter. Logična vrednost, ki jo ta metoda vrača, pa pove, ali predstavitveni dokument sploh ima kazalo.

Lastnosti

Lastnosti se lahko določajo za vsako vozlišče kazala, za vsako stran ali pa za celotno predstavitev. Posameznemu vozlišču kazala (poglavju) lahko poleg številke strani in naslova, ki sta obvezna podatka, dodatne lastnosti nastavimo v metodi `generateOutline()`. Lastnosti posamezne strani se nastavijo v metodi `providePropertiesForSlide()`, celotnega predstavitvenega dokumenta pa v `providePropertiesForPresentation()`. Nastavljamo jih s pomočjo vmesnika `IPropertySource`, ki ga vse te podatkovne strukture izvajajo.

Upodabljanje strani

Za upodabljanje strani ponudnik predstavitev poskrbi v metodi `renderSlide()`. Slika, ki jo vrne ta metoda, mora biti primerek abstraktnega razreda `ImageStorage`. Praviloma izberemo tako izvedbo tega razreda, ki zgolj ovije rezultat knjižnice za uporabljanje strani. Tako se izognemo nepotrebnemu pretvarjanju in kopiranju podatkov, saj ponudnik predstavitev še ne more vedeti, v kakšni obliki bo slika, potrebna za prikaz. O tem sem podrobneje spregovoril že v poglavju 2.4.2.

2.4.4 Urejevalniki za prikaz predstavitev

Izvedbe urejevalnikov za prikaz predstavitev praviloma razširjajo abstraktni razred `PresentationEditor`. Najpreprostejša izvedba je razred `SimplePresentationEditor` iz osnovnega vtičnika. Ta zgolj prikaže stran, povečano na velikost prikazovalnega okna, in omogoča enostavno navigacijo z izbiro številke strani ali pa poglavja v kazalu. Zanimivejša pa je izvedba `CubePresentationEditor` iz vtičnika za prikaz predstavitev, projiciranih na ploskve kocke. O njej bom več govoril v 3. poglavju.

Abstraktni razred `PresentationEditor` razširja razred `EditorPart` iz *Eclipse RCP*, ki priskrbi osnovno funkcionalnost, skupno vsem urejevalnikom. `PresentationEditor` doda podporo za delo s predstavitvenimi dokumenti. Izvedbam urejevalnikov za prikaz predstavitev je tako praviloma potrebno izvesti le šest abstraktnih metod, ki jih prikazuje spodnji primer nefunkcionalne izvedbe (koda 2.1).

Izvedbo metod `createPartControl()` in `setFocus()` potrebuje že `EditorPart`. Prva poskrbi za oblikovanje uporabniškega vmesnika urejevalnika, druga pa pove, kaj naj se zgodi, ko urejevalnik preide v žarišče (angl. *focus*) – običajno preda žarišče glavni komponenti urejevalnika. Metoda `presentationAssigned()` omogoči urejevalniku, da se postavi v začetno stanje, potem ko se mu dodeli pripadajoči predstavitveni dokument (diagram 2.5, dogodek 3). Aktivno predstavitveno stran lahko zamenjamo z metodama

```
1  public class MPE extends PresentationEditor<MyImageStorage> {
2    protected Class<MyImageStorage> getImageStorageClass() {
3      return MyImageStorage.class;
4    }
5    protected void presentationAssigned(IPresentation)
6    protected void selectionChanged(IPresentationSlide)
7    protected void slideUpdated(IPresentationSlide, ImageStorage, boolean)
8    public void createPartControl(Composite)
9    public void setFocus()
10 }
```

Programska koda 2.1: Izvedba urejevalnika za prikaz predstavitev.

`setSelection()` in `selectSlideByNumber()` na razredu `PresentationEditor`. Ti metodi s klicem `selectionChanged()` obvestita urejevalnik, naj osveži prikaz (diagram 2.6, dogodek 1). Kadar se upodobitev neke strani naknadno spremeni, je urejevalnik o tem obveščen z metodo `slideUpdated()` (diagram 2.6, dogodek 18).

Izvedbo metode `getImageStorageClass()` v principu določa že generika v *Javi 1.5*, saj mora vračati razred oblike slik, ki je asociiran z urejevalnikom. Ta omejitev je vsiljena že v času prevajanja kode. Razred oblike slik je potreben pri pretvorbi upodobitve strani v ustrezno obliko, saj se poda kot parameter metode `getAdapter()` na vmesniku `IAdaptable` razreda `ImageStorage` (diagram 2.6, dogodek 15).

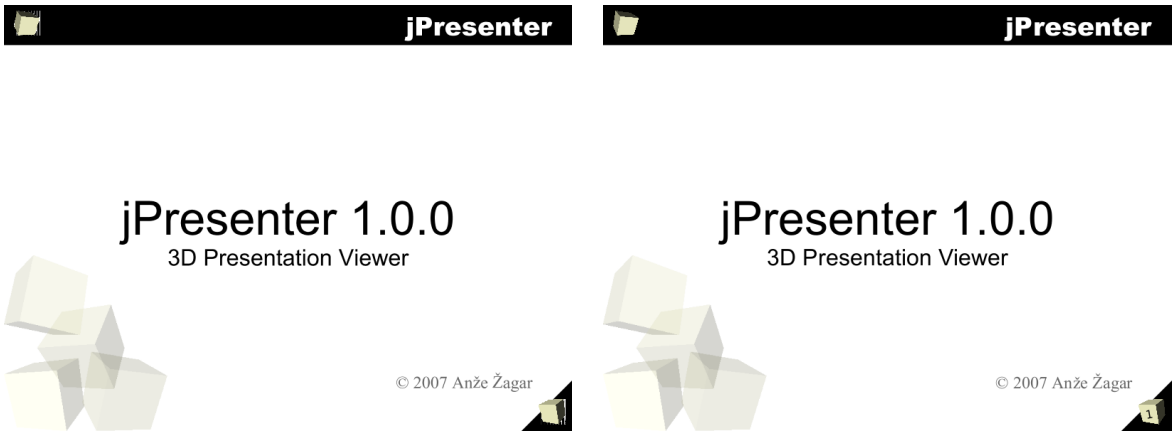
2.5 Prikaz *PDF* dokumentov

Za podporo posameznim predstavitvenim datotečnim zapisom skrbijo posebni vtičniki. Ko enkrat razpolagamo s knjižnicami za razčlenjevanje predstavitvenih dokumentov in za upodabljanje posameznih strani, je izvedba takih vtičnikov zelo enostavna. V tem poglavju se bomo seznanili z dvema knjižnicama, ki razčlenjujeta in upodabljata *PDF* dokumente ter sta uporabljeni v vtičnikih, omenjenih v poglavju 2.2.2.

2.5.1 Izvedba s knjižnico *jPedal*

Knjižnica *jPedal* je narejena za programsko okolje *Java* in deluje tako na *Linux* kot tudi na *Windows* zasnovi. Zasnovana je precej neobjektno, saj je praktično vsa funkcionalnost vključena v razred `PdfDecoder`, ki je hkrati že grafični gradnik za prikaz predstavitve. Posredno razširja razred `JPanel` iz knjižnice `Swing`.

Vtičnik, ki temelji na dotični knjižnici, na začetku ustvari nov razčlenjevalnik (primerek razreda *PdfDecorder*) in mu prepusti, da opravi razčlembo nekega lokalnega *PDF* dokumenta. Razčlenjevalnik se pomni vse dokler za dan dokument obstajajo povezave, ki jih aplikacija vzpostavlja in ruši s pomočjo že omenjenih metod `connect()` in `disconnect()` na ponudniku predstavitev. Kasneje lahko vtičnik po potrebi od razčlenjevalnika zahteva



(a) Knjižnica *jPedal* (b) Knjižnica *Poppler*

Slika 2.7: Primerjava upodobitve strani.

kazalo, splošne podatke o dokumentu ter posameznih straneh ali pa upodobitev posameznih strani. Za podatke o določeni strani ter njeno upodobitev mora razčlenjevalniku predhodno določiti trenutno stran in faktor povečave. Rezultat upodabljanja je slika, predstavljena z razredom `BufferedImage` iz knjižnice *AWT*, kazalo pa se vrne v obliki *XML DOM* dokumenta.

Težava knjižnice *jPedal* je v prvi vrsti počasnost, saj upodabljanje strani lahko vzame tudi nekaj sekund. Drug problem pa je slabo upodabljanje, ki pogosto opazno odstopa od pričakovanih rezultatov. Primer upodobitve neke strani je na sliki 2.7, kjer v primerjavi z rezultati knjižnice *Poppler* spodaj desno manjka številka strani. Prav tako je mogoče opaziti nekatere napake pri izrisu ozadja kock v zgornjem levem ter spodnjem desnem vogalu.

2.5.2 Izvedba s knjižnico *Poppler* preko vmesnika *JNI*

Knjižnica *Poppler* se je razvila iz knjižnice *Xpdf* in je danes uporabljena v večini odprtokodnih *Linux* aplikacij za delo s *PDF* dokumenti. Narejena je v jeziku *C++* in deluje v domačem okolju. Komunikacija znotraj istega procesa (angl. *inprocess*) z okoljem *Java* je možna preko vmesnika *JNI*.

Vtičnik, ki uporablja to knjižnico, zaenkrat deluje le v operacijskem sistemu *Linux* in še to verjetno le na osmi različici distribucije *Fedora*. Knjižnica *Poppler* je namreč dinamično povezana še z drugimi distribucijsko odvisnimi knjižnicami, kot so *FreeType*, *GDK*, *Cairo*, *X11* ... Rešitev za ostale distribucije bi bila statično povezana knjižnica, vključena v sam vtičnik.

Vmesnik *JNI* v okolju *Java* predstavlja skupek statičnih metod, ki jih prikazuje programska koda 2.2. V domačem okolju je na osnovi te kode pripravljena zaglavna (angl. *header*) datoteka za jezik *C* oziroma *C++*, ki je izvedena tako, da ukaze ustrezno delegira

knjižnici *Poppler*. Prevedena knjižnica dobi ime `libjpoppler.so` oziroma `jpoppler.dll` (to ime pričakuje 3. vrstica v kodi 2.2) in mora biti vtičniku dostopna v knjižnični poti (angl. *library path*). Knjižnično pot se praviloma določi celotni aplikaciji ob njenem zagonu, toda to je lahko problematično, saj zagon aplikacije tako postane odvisen od vtičnika. *OSGi* specifikacija zato omogoča, da si vtičnik določi dodatno knjižnično pot do svojih lastnih domačih knjižnic. Da vtičnik deluje neodvisno od ostalih sistemskih namestitev, je smiselno knjižnico tudi statično povezati.

```
1  public class PopplerJNIDelegator {
2    static {
3      System.loadLibrary("jpoppler");
4    }
5    public static synchronized native int openDocument(String uri);
6    public static synchronized native void closeDocument(int doc);
7    public static synchronized native String getOutlineAsXML(int doc);
8    public static synchronized native int pageCount(int doc);
9    public static synchronized native double[] pageSize(int doc, int
         pagenum);
10   public static synchronized native byte[] renderPage(int doc, int
         pagenum, int width, int height, double scale);
11 }
```

Programska koda 2.2: *JNI* vmesnik za knjižnico *Poppler* v okolju *Java*.

Vtičnik ob prvi povezavi z nekim *PDF* dokumentom kliče metodo `openDocument()`. Ta metoda v domačem okolju razčleni dokument ter vrne število, ki je nekakšna ročka na razčlembo in se podaja kot prvi parameter ob klicu ostalih metod. Ko se z dokumentom prekine zadnja povezava, se kliče metoda `closeDocument()`, ki v domačem okolju sprosti vire, dodeljene (angl. *allocated*) za razčlembo.

Knjižnica *Poppler* pripravi kazalo v strukturirani obliki, kar bi bilo nerodno vračati preko vmesnika *JNI*. Zato ga metoda `getOutlineAsXML()` najprej pretvori v tekstovni niz, ki predstavlja *XML* dokument z enako shemo kot v primeru knjižnice *jPedal*. Širina in višina strani v pikslih pri zaslonski resoluciji 72 *dpi* se preko vmesnika *JNI* vrneta kot polje dveh decimalnih števil. Upodobljena stran pa se vrne kot polje bajtov, katerega vsak četverček predstavlja vrednosti rdeče, zelene in modre barvne komponente posameznega piksla ter njegovo alfa vrednost, ki določa prosojnost.

Poglavje 3

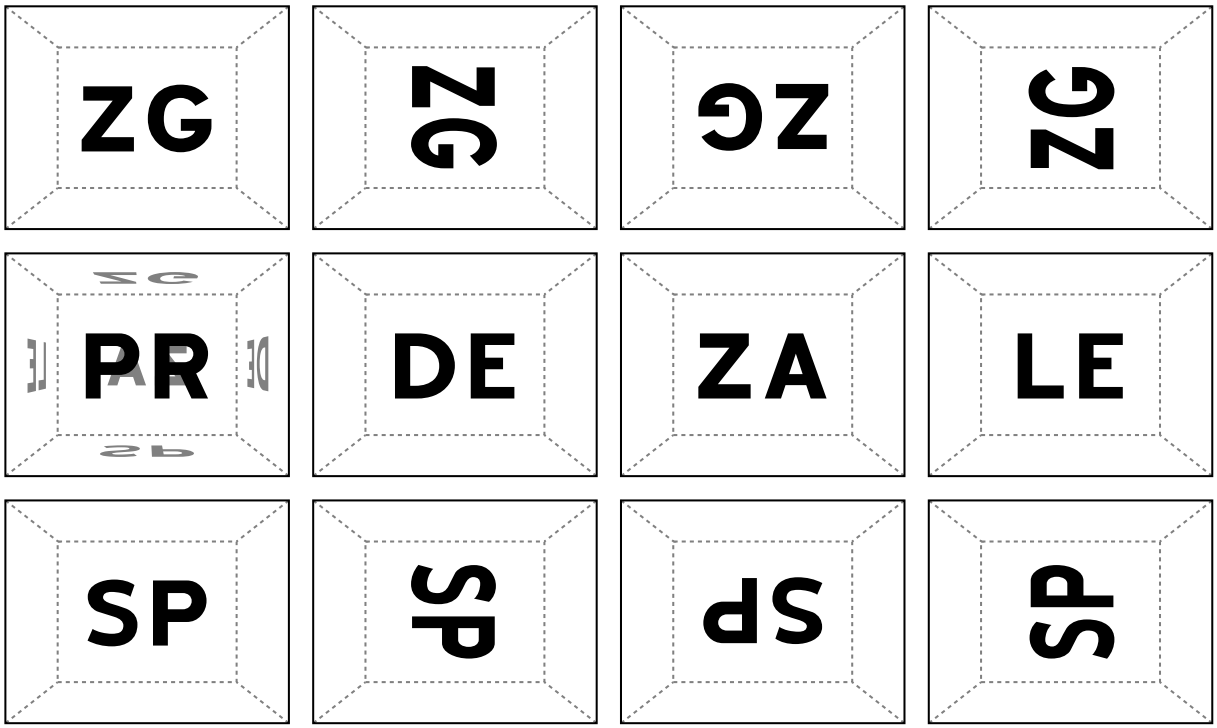
Kocka

V tem poglavju se bom v celoti posvetil delovanju vtičnika za prikaz predstavitev, projiciranih na ploskve kocke (`net.sourceforge.jpresenter.cube`). Na kratko sem ga opisal že v poglavju 2.2.3, njegov način interakcije z ostalo aplikacijo pa je bil razložen v poglavju 2.4.4. Ta vtičnik med drugim vsebuje knjižnico *JOGL*, ki jo koristi za izrisovanje strojno pospešene 3-D grafike. O tem bom več spregovoril v poglavju 3.2. Za opis scene in potek animacije skrbi poseben objektni model, ki opravlja preračunavanje in transformiranje stanj objektov ter njihovo izvrisovanje. Temu objektnemu modelu je posvečeno poglavje 3.3. Na koncu bom govoril še o vrtenju in premikanju kocke z miško ter občutku vztrajnosti kocke, ki povzroči, da je to gibanje bolj zvezno.

3.1 Geometrija kocke

Najprej velja poudariti, da je izraz kocka (angl. *cube*) vzet iz sorodnih in uveljavljenih rešitev, četudi je prikazovano geometrijsko telo lahko poljuben kvader. Po obnašanju in geometrijskih lastnostih naša rešitev oponaša kocko oziroma kvader iz upravitelja oken *Compiz*, ki sem ga omenil že v uvodu. Slika 3.1 prikazuje 12 stabilnih leg, torej leg, v katerih se prednja ploskev kvadra ravno pokrije z dimenzijo prikazovalnega okna (tak primer prikazuje tudi slika 3.2). Vse ostale lege veljajo za nestabilne in če kvader postavimo v eno izmed takih leg, se ta avtomatično zapelje nazaj v najbližjo stabilno lego. Na sliki so ploskve označene s pari začetnih črk: **ZG**ornja, **SP**odnja, **PR**ednja, **DE**sna, **ZA**dnja ter **LE**va. Legi, ko je prednja ploskev obrnjena proti kameri in je na sliki prikazana v srednji vrsti levo, bomo rekli osnovna lega.

Prikazovanju predstavitev so namenjene le lege, prikazane v srednji vrsti (navpične ploskve glede na osnovno lego), zgornja in spodnja ploskev pa sta zaenkrat neuporabljeni in eventuelno rezervirani za prikaz kazala ali pa katerih drugih podatkov o predstavitvi. Vrtenje kvadra je možno okrog osi X in Y, kjer os X teče skozi središči leve in desne, os Y zgornje in spodnje ter os Z prednje in zadnje ploskve. Prav tako je vrtenje okrog osi X omejeno na interval $[-90°, 90°]$. Take omejitve preprečujejo nekatere manj smiselne lege, denimo ko so predstavitvene strani obrnjene na glavo ali pa pod katerim drugim kotom.



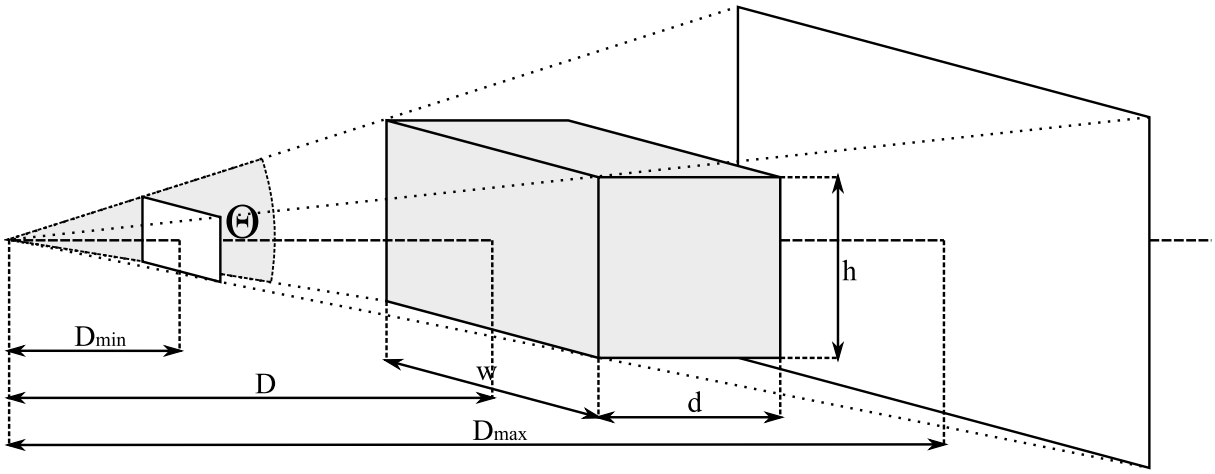
Slika 3.1: 12 stabilnih leg kocke oziroma kvadra.

Stran se zamenja tako, da se kvader obrne okoli osi Y za nek večkratnik pravega kota, dobljena lega pa postane nova osnovna lega.

Vse stabilne lege na sliki 3.1 so pravokotniki v dimenziji prikazovalnega okna. Smiselno se zastavi vprašanje, kakšno geometrijsko telo sploh lahko sestavimo iz takih ploskev. Najbolj praktična rešitev je kvader, ki prilagaja obliko v odvisnosti od njegove orientacije glede na os Y. Ker se za prikaz že tako ali tako uporablja perspektivna projekcija, je to popačenje pri obračanju predstavitvenih strani neprepoznavno. Je pa opazno, če kvader gledamo z vrha, kar se vidi tudi na sliki 3.3.

3.1.1 Volumen gledanja

V našem primeru za izris scene uporabljamo perspektivno projekcijo, kot je prikazano na sliki 3.2. Vidno območje oziroma volumen gledanja (angl. *viewing volume*) pri perspektivni projekciji lahko opišemo s stožcem. Konkretno pri delu s knjižnico *OpenGL* gre običajno za prirezano štiristrano piramido. Poleg stranskih ploskev piramide volumen gledanja omejujeta še ploskvi, ki sta pravokotni na os gledanja in ležita na razdalji D_{min} oziroma D_{max} od kamere. Na sliki sta ti ploskvi prikazani kot bela pravokotnika, obrobljena s polno črto. Razpon gledanja glede na navpično smer je označen kot Θ. Glede na vodoravno smer pa je enak produktu Θr_s, kjer je $r_s = w_s/h_s$ razmerje pogleda (angl. *aspect ratio*) in ga praviloma prilagodimo razmerju širine in višine prikazovalnega okna.



Slika 3.2: Perspektiva gledanja kocke oziroma kvadra.

3.1.2 Velikost in položaj kocke oziroma kvadra

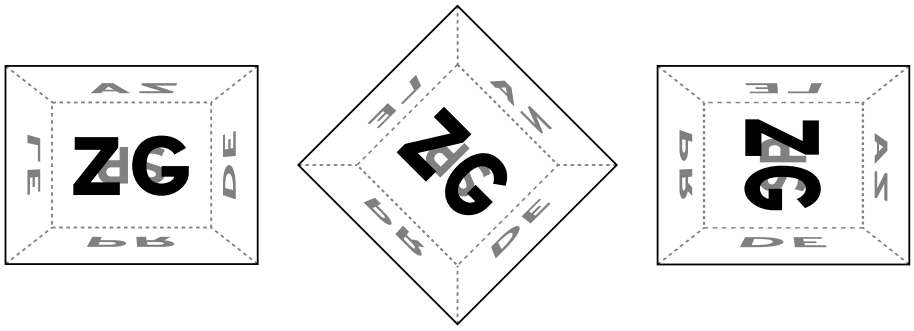
Slika 3.2 prikazuje pogled na kvader v eni izmed njegovih stabilnih leg. Kamero postavimo v točko $(0, 0, 0)$ in jo usmerimo v negativno smer Z osi. S tem zagotovimo, da je pri desnosučnem koordinatnem sistemu X os lahko usmerjena od leve proti desni ter Y os od spodaj navzgor. Za Θ se v aplikaciji *jPresenter* uporablja privzeta vrednost 45°. Ker lahko izberemo poljubno skalo koordinatnega sistema v 3-D prostoru, kvader postavimo vedno tako, da je v stabilni legi središče prednje ploskve postavljeno v točko $(0, 0, -1)$, torej je $D = 1$. Višina kvadra h, ki je hkrati enaka globini d, se izračuna iz enakosti $\tan \frac{\Theta}{2} = \frac{h}{2D}$. Širina se dobi kot produkt višine in razmerja pogleda prikazovalnega okna. Iz povedanega med drugim sledi, da je središče kvadra postavljeno v točko $(0, 0, -D - d/2)$.

Na osnovi opisanih dimenzij kvadra očitno ni problem prikazati prednje, zgornje, zadnje ter spodnje ploskve, saj imajo vse štiri razmerje širine in višine enako prikazovalnemu oknu. Problematični pa sta obe stranski ploskvi (leva in desna), ki sta v bistvu kvadrata. Ob prehajanju med dvema zaporednima predstavitvenima stranema se morata torej dimenziji w in d zamenjati. Prehod želimo, da je zvezen, zato ga računamo kot funkcijo kota vrtenja okrog osi, ki gre skozi središči zgornje in spodnje ploskve kvadra, ϕ_y. Slika 3.3 prikazuje, kako ti dve dimenziji prehajata, če kvader pri obračanju strani opazujemo z vrha oziroma gledamo zgornjo ploskev. Funkciji za izračun novih vrednosti d' in w' imata naslednji obliki:

$$w' = w_s \left(\frac{1}{2} + \frac{1}{2} \cos(2\phi_y) \right) + h_s \left(\frac{1}{2} - \frac{1}{2} \cos(2\phi_y) \right) \tag{3.1}$$

$$d' = w_s \left(\frac{1}{2} - \frac{1}{2} \cos(2\phi_y) \right) + h_s \left(\frac{1}{2} + \frac{1}{2} \cos(2\phi_y) \right) \tag{3.2}$$

Vrednosti w_s in h_s sta širina in višina prednje ploskve v stabilni legi.



Slika 3.3: Vrtenje kocke oziroma kvadra okrog Y osi pri pogledu na zgornjo ploskev.

3.1.3 Povečava

Ker uporabljamo perspektivno projekcijo, lahko predstavitveno stran povečamo oziroma pomanjšamo na dva načina. Prvi način, ki deluje tudi za ortografske projekcije, je spreminjanje dimenzij kvadra proporcionalno z želenim faktorjem povečave p. Pri perspektivni projekciji je potrebno zagotoviti še, da prednja ploskev ostane na konstantni oddaljenosti od kamere, kar pomeni tudi premikanje središča kvadra. Če hočemo kvader vendarle obdržati na istem mestu, pa moramo upoštevati še faktor povečave $p_p = D/D'$, ki ga povzroči perspektiva zaradi približanja prednje ploskve na oddaljenost D' od kamere. Iščemo torej tak faktor povečave kvadra p_d, da bo veljalo $p = p_d \cdot p_p$.

Središče kvadra torej ohranimo na istem mestu in izpeljemo D':

$$D + \frac{d}{2} = D' + \frac{d'}{2} \qquad , \qquad d' = d \cdot p_d \qquad \Rightarrow \qquad D' = D + \frac{d}{2} - \frac{d \cdot p_d}{2}$$

In izračunamo p_d:

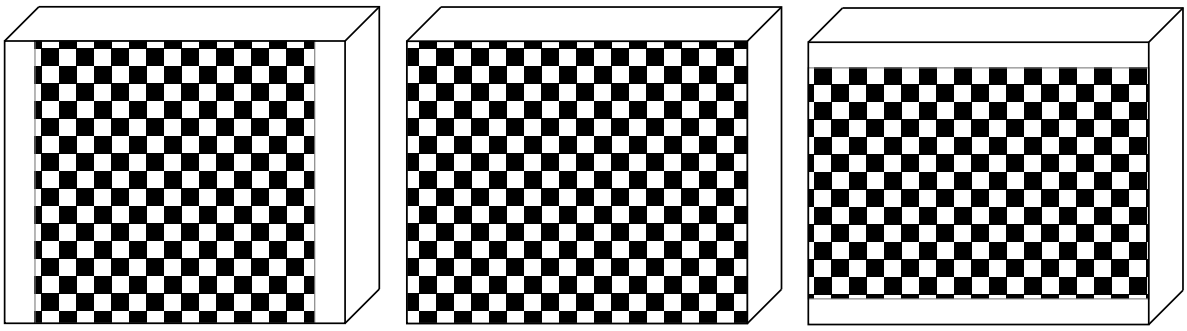
$$\begin{aligned} p = p_d \cdot p_p = p_d \frac{D}{D'} &= \frac{D p_d}{D + \frac{d}{2}(1 - p_d)} \\ p\left(D + \frac{d}{2}\right) &= p_d\left(D + \frac{p \cdot d}{2}\right) \\ p_d &= \frac{D + \frac{d}{2}}{D + \frac{p \cdot d}{2}} \cdot p \end{aligned} \tag{3.3}$$

Opisana rešitev je praktična, kadar želimo podpreti tako perspektivno kot tudi ortografsko projekcijo. Preprostejši način, ki pa je možen izključno pri perspektivni projekciji, pa zgolj s približevanjem in oddaljevanjem kvadra izkorišča učinek perspektive $p = p_p = D/D'$.

V praktični izvedbi povečavo običajno korakoma povečujemo oziroma pomanjšujemo, na primer z obračanjem koleščka na miški. Če je n število premikov koleščka glede na izhodiščno lego, skupen faktor povečave izračunamo po enačbi $p = p_1^n$, kjer je p_1 aplikacijsko določen faktor povečave enega koraka.

3.1.4 Prikaz predstavitvenih strani

Predstavitvene strani se kot teksture lepijo na ploskve kvadra. Velikost teksture prilagodimo tako, da se optimalno ujema z velikostjo prednje ploskve v osnovni legi ($w_s \times h_s$). Pod besedno zvezo "optimalno ujemanje" razumemo maksimalno velikost, pri kateri je prikazana še celotna in nepopačena tekstura. Razmerje širine in višine teksture (r_t) je seveda lahko različno od razmerja $r_s = w_s/h_s$.



(a) $r_s > r_t$ (b) $r_s = r_t$ (c) $r_s < r_t$

Slika 3.4: Lepljenje teksture na ploskev kvadra.

Ločimo tri primere, ki jih prikazuje slika 3.4. Splošno se relativne koordinate ploskve preslikajo v relativne koordinate teksture po enačbah:

(a) $r_s > r_t$: $x' = \left(x - \frac{1}{2}\right)\frac{r_s}{r_t} + \frac{1}{2}$, $y' = y$

(b) $r_s = r_t$: $x' = x$, $y' = y$

(c) $r_s < r_t$: $x' = x$, $y' = \left(y - \frac{1}{2}\right)\frac{r_t}{r_s} + \frac{1}{2}$

Iz tabele 3.1 lahko odčitamo, kako se preslikajo oglišča ploskve, kar bom potreboval v poglavju 3.2 za določitev teksture s knjižnico *OpenGL*. Vse vrednosti so izven intervala $(0, 1)$, saj se robovi ploskve bodisi pokrijejo z robovi teksture ali pa so izven nje.

primer	levo	zgoraj	desno	spodaj
(a)	$\frac{1}{2}(1 - \frac{r_s}{r_t})$	0	$\frac{1}{2}(1 + \frac{r_s}{r_t})$	1
(b)	0	0	1	1
(c)	0	$\frac{1}{2}(1 - \frac{r_t}{r_s})$	1	$\frac{1}{2}(1 + \frac{r_t}{r_s})$

Tabela 3.1: Relativne koordinate teksture v robovih ploskve pri optimalnem prileganju.

3.2 Upodabljanje kocke

3-D grafika se v aplikaciji *jPresenter* upodablja z uporabo knjižnice *JOGL*. *JOGL* v bistvu ni nič drugega kot ovojnica za knjižnico *OpenGL*, ki preko vmesnika *JNI* delegira ukaze

OpenGL v domače okolje. V tem poglavju bom nakazal [3], kako se uporabi omenjeno knjižnico za izris kocke, opisane v poglavju 3.1. Za bolj poglobljeno razumevanje o delu s knjižnico *OpenGL* pa priporočam branje druge literature, na primer [17].

3.2.1 Prikazovanje vsebine *OpenGL* v *Eclipse RCP*

Knjižnica *JOGL* za prikaz vsebine *OpenGL* nudi razreda `GLCanvas` ter `GLJPanel`. Prvi je razširitev razreda `Canvas` iz knjižnice *AWT*, drugi pa razširja `JPanel` iz knjižnice *Swing*. *Eclipse RCP* priskrbi svojo izvedbo tovrstnega prikazovalnika, ki mu je prav tako ime `GLCanvas`, le da temelji na knjižnici *SWT*. Gre za običajno komponento grafičnega vmesnika, ki ji naredimo *GL* kontekst in ji preko tega konteksta izrišemo 3-D vsebino.

Postopek priprave komponente `GLCanvas` ter njenega *GL* konteksta je nakazan v programski kodi 3.1. Ustvarjanje konteksta (vrstica 5) zahteva, da se predhodno določi trenutno prikazovalno komponento (vrstica 4), saj se na tak način vzpostavi povezava med obema. Do konteksta sme v danem trenutku dostopati le tista nit, ki si ga predhodno prisvoji (vrstica 6). Da si ga lahko kasneje prisvoji katera druga nit, je potrebno kontekst po uporabi vedno sprostiti (vrstica 8). Izkaže se, da se morajo v *Eclipse RCP* vse operacije nad *GL* kontekstom vršiti znotraj niti uporabniškega vmesnika, saj sicer lahko prihaja do napak.

V vrstici 2 omogočimo še dvojno medpomnjenje (angl. *double buffering*), s katerim lahko vsako naslednjo sliko najprej pripravimo v posebnem medpomnilniku in jo šele nato prikažemo s klicem `swapBuffers()` na komponenti `GLCanvas`. To preprečuje utripanje slike, do katerega bi prihajalo, če bi izrisovali neposredno na ekran.

```
1 GLData data = new GLData();
2 data.doubleBuffer = true;
3 canvas = new GLCanvas(parent, SWT.NONE, data);
4 canvas.setCurrent();
5 context = GLDrawableFactory.getFactory().createExternalGLContext();
6 context.makeCurrent();
7 initialize(context.getGL());
8 context.release();
```

Programska koda 3.1: Priprava komponente `GLCanvas` ter njenega *GL* konteksta.

Funkcija `initialize()` (vrstica 7) poskrbi za začetne nastavitve *GL* konteksta ter nastavitve, ki se tekom izvajanja ne bodo več bistveno spreminjale. V podrobnosti, kaj vse bi ta funkcija lahko počela, se tu ne bom spuščal.

3.2.2 Nastavljanje projekcije

Projekcijo je potrebno nastaviti ob vsaki spremembi velikosti prikazovalne komponente. Postopek je prikazan v programski kodi 3.2. Najprej določimo izrisno okno (angl. *vie-*

wport), ki želimo, da zaobsega celotno prikazovalno komponento (vrtica 1). w_w in h_w predstavljata zaslonsko velikost (širino in višino) te komponente. Nato preklopimo v projekcijski matrični način (vrstica 2), da določimo projekcijsko transformacijo kamere. Določimo jo tako, da projekcijsko matriko najprej nastavimo na identiteto (vrstica 3) in jo zmnožimo s perspektivno matriko (vrstica 5). Parametre funkcije `gluPerspective()` sem razložil že v poglavju 3.1.1. Preden nadaljujemo z risanjem scene, preklopimo nazaj v oblikovni matrični način (vrstica 6).

```
1 gl.glViewport(0, 0, w_w, h_w);
2 gl.glMatrixMode(GL.GL_PROJECTION);
3 gl.glLoadIdentity();
4 GLU glu = new GLU();
5 glu.gluPerspective(Θ, w_s/h_s==w_w/h_w, D_min, D_max);
6 gl.glMatrixMode(GL.GL_MODELVIEW);
```

Programska koda 3.2: Nastavljanje projekcije v *JOGL*.

3.2.3 Prikaz kocke

Programska koda 3.3 izriše prednjo ploskev kocke oziroma kvadra, opisanega v poglavju 3.1. Najprej določimo velikost osnovne ploskve (vrstici 2 in 3), kot je opisano v poglavju 3.1.2. Razmerje pogleda osnovne ploskve (vrstica 1) vzamemo, da je enako razmerju pogleda prikazovalne komponente. Širino in globino kocke (vrstici 5 in 7) izračunamo po enačbah 3.1 in 3.2. Za vpenjanje teksture na ploskev potrebujemo še spremenljivke `levo`, `zgoraj`, `desno` in `spodaj` (vrstice 8–11), ki so izračunane po tabeli 3.1. Povečavo upoštevamo s premikom za ${p_1}^n$ nazaj po Z osi koordinatnega sistema (vrstica 13) v skladu s povedanim v poglavju 3.1.3.

V vrstici 12 najprej nastavimo transformacijsko matriko na identiteto. To matriko nato v naslednji vrstici zmnožimo z matriko translacije, ki prestavi koordinatni sistem na mesto, kjer želimo središče kocke. Sledi vrtenje okrog X (vrstica 14) ter nato še okrog Y osi (vrstica 15). V začetku poglavja 3.1 sem omenil, da za naš primer vrtenje okrog Z osi ni smiselno. Sedaj, ko smo se postavili v koordinatni sistem kocke, lahko izrišemo njene ploskve. Prednja ploskev se izriše med vrsticama 17 in 26. Konstanta `GL_QUADS` pove, da površino opisujemo s štirikotniki. Posamezno oglišče štirikotnika oziroma ploskve določimo z ukazom `glVertec3f` tako, da mu podamo koordinate v kockinem koordinatnem sistemu. Predhoden ukaz `glTexCoord2f` pa za dano oglišče pove, na katero točko na teksturi (lahko tudi izven nje) naj bo vpeto. Teksturo `tid`, ki jo vpenjamo na ploskev, določimo v vrstici 16. O tem bom več povedal v poglavju 3.2.4.

```
1  r_s = r_w = w_w/h_w;
2  h_s = 2.0*tan(0.5*Θ);
3  w_s = r_s*h_s;
4  state = 0.5 − 0.5*cos(2.0*φ_y);
5  w = (1.0 − state)*w_s + state*h_s;
6  h = h_s;
7  d = state*w_s + (1.0 − state)*h_s;
8  levo = r_s > r_t ? 0.5*(1.0 − r_s/r_t) : 0.0;
9  zgoraj = r_s < r_t ? 0.5*(1.0 − r_t/r_s) : 0.0;
10 desno = r_s > r_t ? 0.5*(1.0 + r_s/r_t) : 1.0;
11 spodaj = r_s < r_t ? 0.5*(1.0 + r_t/r_s) : 1.0;
12 gl.glLoadIdentity();
13 gl.glTranslatef(0.0, 0.0, −0.5*h_s − p_1^n);
14 gl.glRotatef(φ_x, 1.0, 0.0, 0.0);
15 gl.glRotatef(φ_y, 0.0, 1.0, 0.0);
16 gl.glBindTexture(GL.GL_TEXTURE_RECTANGLE_ARB, tid);
17 gl.glBegin(GL.GL_QUADS);
18   gl.glTexCoord2f(levo*w_t, zgoraj*h_t);
19   gl.glVertex3f(−0.5*w, 0.5*h, 0.5*d);
20   gl.glTexCoord2f(desno*w_t, zgoraj*h_t);
21   gl.glVertex3f(0.5*w, 0.5*h, 0.5*d);
22   gl.glTexCoord2f(desno*w_t, spodaj*h_t);
23   gl.glVertex3f(0.5*w, −0.5*h, 0.5*d);
24   gl.glTexCoord2f(levo*w_t, spodaj*h_t);
25   gl.glVertex3f(−0.5*w, −0.5*h, 0.5*d);
26 gl.glEnd();
```

Programska koda 3.3: Prikaz prednje ploskve s knjižnico *JOGL*.

3.2.4 Prosojnost in zlivanje tekstur

Pri določanju barve nekega gradnika lahko poleg treh osnovnih barvnih komponent (rdeče, modre in zelene) podamo še tako imenovano *alfa* vrednost, ki predstavlja stopnjo prosojnosti (angl. *transparency*). Prav tako imajo teksture lahko poseben alfa sloj, ki določa prosojnost vsake pike posebej.

Upodabljanje prosojnih gradnikov v 3-D prostoru prinese tudi nekatere dodatne težave. V splošnem je pri izrisovanju scene pomembna globina položaja posameznih gradnikov, saj lahko tisti, ki so bližje kameri, prekrivajo ostale. Za pravilen izris moramo zato izrisovati gradnike v vrstnem redu od najbolj oddaljenega proti najbližjemu. V ta namen obstaja precej algoritmov za razvrščanje gradnikov, vendar nobeden ni enostaven za izvedbo. Vseeno pa lahko problem zaobidemo z uporabo globinskega testa, ki povzroči, da knjižnica *OpenGL* za vsako točko rasterske upodobitve poišče ter izriše le najbližji gradnik v tisti smeri. To pa deluje le, če ta gradnik ni prosojen. Na srečo je razvrščanje ploskev pri kocki zelo enostavno. Aplikacija *jPresenter* jih razvrsti po razliki v kotu med njihovimi normalami ter normalo prednje ploskve v stabilni legi.

```
1  int TUNIT[] = new int[] { GL.GL_TEXTURE0, GL.GL_TEXTURE1, ... };
2  int tid[] = new int[M];
3  gl.glGenTextures(M, tid, 0);
4  for (int i = 0; i < M; i++); {
5    gl.glActiveTexture(TUNIT[i]);
6    gl.glBindTexture(GL.GL_TEXTURE_RECTANGLE_ARB, tid[i]);
7    gl.glTexImage2D(GL.GL_TEXTURE_RECTANGLE_ARB, 0, GL.GL_RGBA, w_t, h_t, 0,
       GL.GL_RGBA, GL.GL_UNSIGNED_BYTE, tdata[i]);
8    gl.glTexParameteri(GL.GL_TEXTURE_RECTANGLE_ARB, GL.GL_TEXTURE_WRAP_S,
       GL.GL_CLAMP_TO_BORDER);
9    gl.glTexParameteri(GL.GL_TEXTURE_RECTANGLE_ARB, GL.GL_TEXTURE_WRAP_T,
       GL.GL_CLAMP_TO_BORDER);
10
11   double CONSTANT = new double[] { 0.9, 0.9, 0.9, 0.5 };
12   gl.glTexEnvdv(GL.GL_TEXTURE_ENV, GL.GL_TEXTURE_ENV_COLOR, CONSTANT, 0);
13   gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_TEXTURE_ENV_MODE, GL.GL_COMBINE);
14   gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_COMBINE_RGB, GL.GL_INTERPOLATE);
15   gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_SOURCE1_RGB, GL.GL_PREVIOUS);
16   gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_OPERAND0_RGB, GL.GL_SRC_COLOR);
17   gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_SOURCE0_RGB, GL.GL_TEXTURE);
18   gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_OPERAND1_RGB, GL.GL_SRC_COLOR);
19   gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_SOURCE2_RGB, GL.GL_CONSTANT);
20   gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_OPERAND2_RGB, GL.GL_SRC_COLOR);
21   gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_COMBINE_ALPHA, GL.GL_REPLACE);
22   gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_SOURCE0_ALPHA, GL.GL_CONSTANT);
23   gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_OPERAND0_ALPHA, GL.GL_SRC_ALPHA);
24
25   gl.glEnable(GL.GL_TEXTURE_RECTANGLE_ARB);
26 }
27 ...
28 gl.glBegin(GL.GL_QUADS);
29   for (i = 0; i < M; i++) gl.glMultiTexCoord2d(TUNIT[i], levo, zgoraj);
30   gl.glVertex3f(-0.5*w, 0.5*h, 0.5*d);
31   for (i = 0; i < M; i++) gl.glMultiTexCoord2d(TUNIT[i], desno, zgoraj);
32   gl.glVertex3f(0.5*w, 0.5*h, 0.5*d);
33   for (i = 0; i < M; i++) gl.glMultiTexCoord2d(TUNIT[i], desno, spodaj);
34   gl.glVertex3f(0.5*w, -0.5*h, 0.5*d);
35   for (i = 0; i < M; i++) gl.glMultiTexCoord2d(TUNIT[i], levo, spodaj);
36   gl.glVertex3f(-0.5*w, -0.5*h, 0.5*d);
37 gl.glEnd();
```

Programska koda 3.4: Zlivanje tekstur na ploskev.

Vpenjanje teksture sem opisal že na primeru v programski kodi 3.3. Konstanta `GL_TEXTURE_RECTANGLE_ARB` v vrstici 16 pove, da je tekstura podana z uporabo *OpenGL* razširitve *ARB_texture_rectangle*. Za razliko od standardnih tekstur (`GL_TEXTURE_2D`) so te teksture lahko poljubnih dimenzij in širina ter višina nista nujno potenci števila 2. Po drugi strani ta razširitev ni nujno povsod podprta, zato se velja pred uporabo o tem prepričati z ukazom `isExtensionAvailable()`, ki je del knjižnice *JOGL*.

Na isto ploskev lahko vpnemo tudi več tekstur in jim določimo praktično poljuben način zlivanja. To omogoča *OpenGL* razširitev *GL_ARB_multitexture*. Maksimalno število tekstur, ki jih lahko vpnemo na eno ploskev, je odvisno od števila razpoložljivih teksturirnih enot. Ugotovimo ga lahko z ukazom `glGetIntegerv(GL_MAX_TEXTURE_UNITS, val, 0)`.

Programska koda 3.4 prikazuje primer zlivanja M tekstur, opisanih v polju `tdata`. Polje `TUNIT` (vrstica 1) vsebuje identifikatorje teksturirnih enot. Posamezno teksturirno enoto aktiviramo v vrstici 5 in s tem povemo, da se nanjo navezujejo vse nadaljnje operacije. V vrstici 6 ji izberemo teksturo, ki jo v naslednji vrstici naložimo v videopomnilnik z ukazom `glTexImage2D()`. Vrstici 8 in 9 določata, kako naj se tekstura nadaljuje na tistih delih ploskve, ki jih eksplicitno ne prekriva.

Način zlivanja tekstur določajo vrstice 11–23. V konkretnem primeru se barvne komponente interpolirajo (vrstica 14) od prejšnjih vrednosti (vrstici 15 in 16) do vrednosti trenutne teksture (vrstici 17 in 18) za konstanten faktor 0.9 (vrstice 11, 19 in 20). Alfa vrednosti pa se nadomestijo (vrstica 21) za konstantno vrednost 0.5 (vrstice 11, 22 in 23).

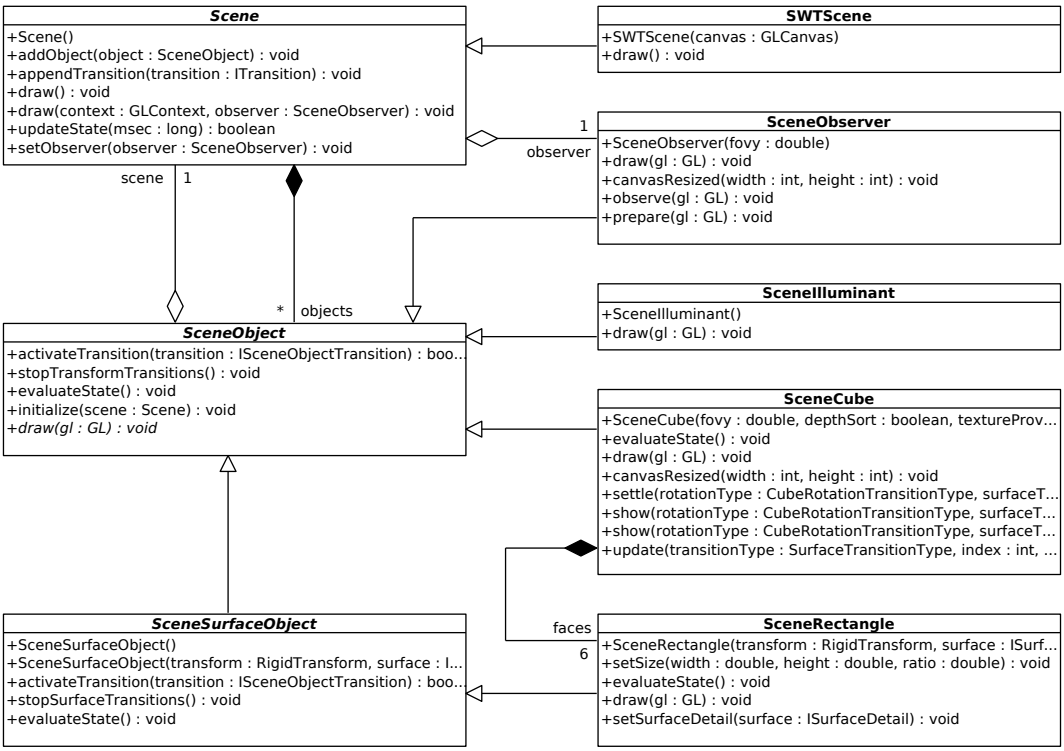
Vrstice 28–37 v bistvu nadomeščajo vrstice 17–26 v programski kodi 3.3 tako, da na posamezno oglišče ploskve vpnejo vseh M tekstur, določenih (vrstica 6) posameznim teksturirnim enotam.

3.3 Opis scene in animacije

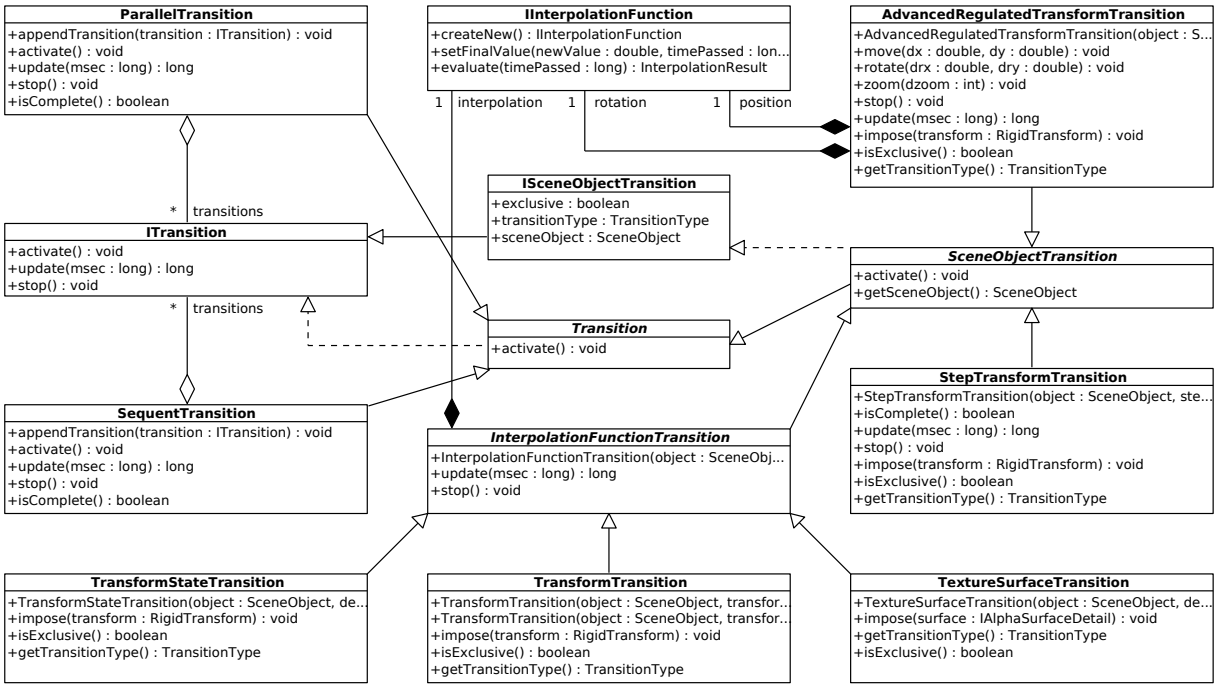
V poglavju 3.2 smo si na kratko ogledali, kako bi zastavljen problem rešili neposredno z uporabo knjižnice *JOGL*. Tak proceduralen pristop je precej tog in neroden za upodabljanje kompleksnejše grafike, zato bom sedaj vpeljal bolj intuitiven objekten model za opis 3-D scene in animacije. Že v uvodu (poglavje 1.4.3) je bilo govora o visokonivojskih programskih vmesnikih za delo s 3-D grafiko ter o tako imenovanih grafih scen. Nekaj podobnega sem razvil za aplikacijo *jPresenter*.

3.3.1 Scena

Diagram razredov na sliki 3.5 prikazuje razrede za opis scene. Abstraktni razred `Scene` predstavlja celotno sceno, ki se s funkcijo `draw()` izriše bodisi v podan ali pa privzet *GL* kontekst. Za privzet kontekst mora poskrbeti razširitev tega razreda, ki praviloma vsebuje komponento grafičnega vmesnika za izris 3-D vsebine (npr. razred `SWTScene` vsebuje komponento `GLCanvas` iz knjižnice *SWT* – poglavje 3.2.1). To tesno združevanje scene in grafične komponente oziroma njenega *GL* konteksta je smiselno, kadar je sama vsebina



Slika 3.5: *UML* diagram razredov scene.



Slika 3.6: *UML* diagram razredov prehajanja.

scene odvisna od velikosti prikazovalnika, kot sta v našem primeru denimo dimenzija kocke in ločljivost tekstur.

Omenjena funkcija `draw()` izriše trenutno stanje. Stanje se mora predhodno osvežiti glede na čas, ki je pretekel od zadnje osvežitve. Za to poskrbi funkcija `updateState()`, ki jo iz posebne niti, tempirane na določen časovni interval, kliče primerek razreda `Player`. Povratna logična vrednost te funkcije pove, ali se je karkoli spremenilo. Le v primeru, če je do spremembe prišlo, se nato asinhrono iz niti uporabniškega vmesnika izvede funkcija `draw()`. Obe funkciji morata biti sinhronizirani, da preprečimo izrisovanje nekonsistentnih oziroma deloma osveženih stanj.

Sceni s klicanjem funkcije `addObject()` dodajamo objekte, ki so izvedbe abstraktnega razreda `SceneObject`. Tovrstni objekti so lahko kamere (`SceneObserver`), svetila (`SceneIlluminant`) ali pa konkretni objekti v prostoru. Nadalje so lahko hierarhično sestavljeni iz drugih podobjektov (npr. kocka `SceneCube` je sestavljena iz šestih ploskev `SceneRectangle`). Vsak objekt ima določen položaj in orientacijo v prostoru, za svoj izris pa poskrbi v funkciji `draw()`, ki jo rekurzivno po celotnem grafu scene izvaja prej omenjena istoimenska funkcija razreda `Scene`. Položaj in orientacija sta dve izmed lastnosti stanja objekta, ki ju skozi tok animacije spreminjajo primerki razredov prehajanja (diagram 3.6). Med ostale tovrstne lastnosti štejemo še stopnjo prosojnosti površine na objektih, ki so primerki razširitve abstraktnega razreda `SceneSurfaceObject`.

3.3.2 Prehajanje stanj

Kot je bilo rečeno, za animacijo skrbijo razredi prehajanja (angl. *transition*), ki so prikazani na diagramu 3.6. V osnovi vsi razširajo abstraktni razred `Transition` in izvajajo vmesnik `ITransition`. Posamezen primerek razreda prehajanja je bodisi vsebovalnik drugih primerkov ali pa povzroča prehajanje stanja nekega točno določenega objekta v sceni. Vsebovalnika sta denimo razreda `ParallelTransition` in `SequentTransition`, ki vzporedno oziroma zaporedno izvajata vsebovane primerke razredov prehajanja.

Prehajanje stanja določenega objekta v sceni povzročajo tisti razredi prehajanja, ki izvajajo vmesnik `ISceneObjectTransition` oziroma razširajo abstraktni razred `SceneObjectTransition`. Nekateri si pri preračunavanju vmesnih stanj pomagajo z interpolacijskimi funkcijami (poglavje 3.3.3) in praviloma razširjajo abstraktni razred `InterpolationFunctionTransition`. Interpolacijska funkcija na osnovi časa, ki je pretekel od aktivacije primerka razreda prehajanja, izračuna pot opravljenega prehoda. Različne izvedbe interpolacijske funkcije omogočajo različno dinamiko prehajanja.

Prehajanje se lahko navezuje na prostorsko stanje objekta (položaj in orientacija v prostoru), ki je opisano z razredom `RigidTransform`, ali pa na nekatere druge lastnosti, kot je na primer stopnja prosojnosti, stopnja prelivanja dveh tekstur (angl. *cross dissolve*) ... Primerek razreda prehajanja se prične izvajati na pripadajočem objektu ob klicu funkcije `activate()`, kar se zgodi, ko je dodan v sceno s funkcijo `appendTransition()`, ali pa ko pride na vrsto za izvajanje v primerkih prej omenjenih vsebovalnih razredov prehajanja `ParallelTransition` in `SequentTransition`.

Postopek osveževanja prostorskih stanj objektov v sceni je prikazan z algoritmom 3.1. Trenutna stanja posameznega objekta se vedno izračunavajo glede na njegovo začetno stanje. Če je na danem objektu določenih več aktivnih primerkov prehajanja, se ti ob vsaki osvežitvi, v enakem vrstnem redu, kot so bili aktivirani, eden za drugim (zanka v koraku 6) odrazijo na stanju T – ga transformirajo (korak 8). Stanje T je prednastavljeno na identiteto (korak 5). Prehajanje P je v prvi iteraciji zanke (korak 6) vedno primerek *končanega* prehajanja, ki predstavlja začetno stanje objekta (komentar 7).

Ker se tudi ostala prehajanja lahko enkrat končajo, jih je smiselno nekako izključiti iz postopka vsakokratnega osveževanja, sicer se z aktiviranjem novih prehajanj dolžina zanke le povečuje. Če bi bile transformacije komutativne, bi lahko vsako zaključeno prehajanje kar izbrisali in ustrezno popravili začetno stanje – recimo temu, da bi zaključeno prehajanje *združili* v prehajanje, ki predstavlja začetno stanje. Žal temu ni tako, zato pa lahko izkoristimo asociativnost transformacij, kar nam omogoča, da združujemo *zaporedna* končana prehajanja (koraki 10–14).

Kot dodatno optimizacijo izkoristimo še lastnost nekaterih izvedb razredov prehajanja. Prehajanje namreč v svojem končanem stanju prostorsko stanje T, bodisi popolnoma povozi (popravljeno stanje $T' = P(T)$ ni odvisno T) bodisi ga le prilagodi (T' je odvisno od T). V prvem primeru lahko odmislimo vse primerke prihajanj, ki se odražajo na T pred tem primerkom, stanje tega primerka pa določimo za novo začetno stanje objekta (korak 11).

```
 1: za vsak aktiven primerek prehajanja P :
 2:    osveži trenutno stanje prehajanja P za dan čas od prejšnje osvežitve Δt
 3: konec za
 4: za vsak objekt O v sceni :
 5:    T ← I
 6:    za vsak aktiven primerek prehajanja P objekta O :
 7:       {prvi primerek P vedno predstavlja začetno stanje objekta O}
 8:       T ← P(T) {prehajanje P odrazi svoje trenutno stanje na prostorskem stanju T}
 9:       če prehajanje P je končano potem
10:          če P(T) je neodvisno od T {P v koraku 8 povozi T} potem
11:             zbriši vsa predhodna prehajanja {tudi če niso končana}
12:          sicer če prejšnje prehajanje P' je končano potem
13:             združi P' in P {v tem vrstnem redu!}
14:          konec če
15:       konec če
16:       P' ← P
17:    konec za
18: konec za
```

Algoritem 3.1: Osveževanje prostorskih stanj objektov.

Prehajanje prostorskega stanja po korakih

Gre za najenostavnejšo obliko prehajanja, ki ne upošteva časa od zadnje osvežitve in se zato zanaša na točnost osveževanja animacije. Izvedena je v razredu `StepTransformTransition`. Nad prostorskim stanjem objekta N-krat odrazi podano prostorsko transformacijo, pri čemer je N število osvežitev oziroma korakov od začetka izvajanja.

Postopno prehajanje v novo prostorsko stanje

Pri tej obliki prehajanja se ob vsaki osvežitvi prostorsko stanje objekta interpolira (dodatek A) v podano ciljno stanje za delež, ki ga ugotovi s pomočjo podane interpolacijske funkcije. Izvaja jo razred `TransformStateTransition`. Ko je tako prehajanje gotovo, vsa ostala prehajanja, ki se vršijo pred njim, izgubijo vpliv in jih lahko pobrišemo (korak 11 algoritma 3.1).

Postopno prehajanje za dano spremembo prostorskega stanja

Ta oblika je izvedena v razredu `TransformTransition`. Z vsako osvežitvijo prostorskega stanja objekta se to transformira za izračun interpolacije med identiteto ter podano transformacijo (dodatek A). Delež interpolacije ugotovi s pomočjo podane interpolacijske funkcije. Ko je prehajanje gotovo, se lahko združi s sosednjimi končanimi prehajanji (korak 13 algoritma 3.1).

Uporabniško vodeno prehajanje stanja

Uporabniško vodeno prehajanje stanja je namenjeno predvsem navigaciji objektov s pomočjo miške (poglavje 3.4.2). Izvedeno je v razredu `AdvancedRegulatedTransformTransition`. Smisel tega prehajanja je, da objekt (orientacija in položaj) sledi premikom miške, vendar hkrati omogoča upoštevanje določene vztrajnosti, s čimer zgladi njegovo gibanje tudi v primeru naglih premikov miške. Za preračunavanje vmesnih stanj si pomaga z ravninskimi interpolacijskimi funkcijami (poglavje 3.3.4), ki se morajo znati tudi dinamično prilagajati spreminjanju ciljne lokacije.

3.3.3 Interpolacijske funkcije

Interpolacijska funkcija je funkcija poti opravljenega prehoda v odvisnosti od pretečenega časa t. Definirana je v skladu s preslikavo 3.4. Notranje stanje ji med drugim določa končna pot s, na osnovi katere je možno izračunati delež opravljenega prehoda.

$$s(t) : [0, T] \rightarrow \Re \;\; ; \;\; s(0) = 0, s(T) = s \tag{3.4}$$

Vse interpolacijske funkcije izvajajo vmesnik `IInterpolationFunction`. Funkcija `evaluate()` za podan tenutek t vrne strukturo `InterpolationResult`, ki poleg opravljene poti pove še, ali je bilo prehajanje končano, ter čas, ki je v tem primeru ostal.

Opozoriti velja, da to, da je pot enaka končni poti, še ne pomeni nujno, da je prehajanje končano. Prav tako lahko pot vmes tudi prekorači končno pot, kot v primeru dušene nihajoče interpolacijske funkcije (graf 3.8(f)).

Dani interpolacijski funkciji lahko končno pot naknadno spremenimo s funkcijo `setFinalValue()`. Pri tem poleg nove vrednosti končne poti podamo še čas, ob katerem naj bi do spremembe prišlo, da funkcija po potrebi ustrezno prilagodi svoje notranje stanje. Nekatere interpolacijske funkcije namreč potrebujejo ta čas, da lahko ohranijo zveznost poti v odvisnosti od časa $s(t)$.

Takojšnja interpolacijska funkcija

Takojšna interpolacijska funkcija (graf 3.8(a)) do podanega časa T vrača 0, nato pa se v trenutku konča in prične vračati končno pot s. Ni težko ugotoviti, da funkcija poti $s(t)$ ni zvezna.

Zakasnjena interpolacijska funkcija

Podobno kot takojšnja funkcija čaka podan čas T, le da nato prične izvajati drugo interpolacijsko funkcijo.

Linearna interpolacijska funkcija

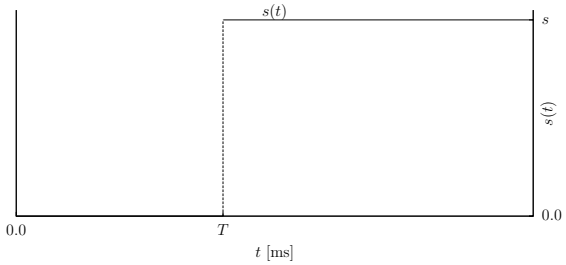
Linearna interpolacijska funkcija (graf 3.8(b)) s podano konstantno hitrostjo v linearno povečuje pot $s(t)$, dokler ta ne pride do končne poti s. Pot $s(t)$ je zvezna C^0, saj ima že prvi odvod $v(t)$ točki nezveznosti na začetku ter koncu intervala $[0, T = s/v]$. Za ohranjanje zveznosti C^0 se pri spremembi končne poti s tej interpolacijski funkciji ni potrebno posebej prilagajati.

Pospešena interpolacijska funkcija

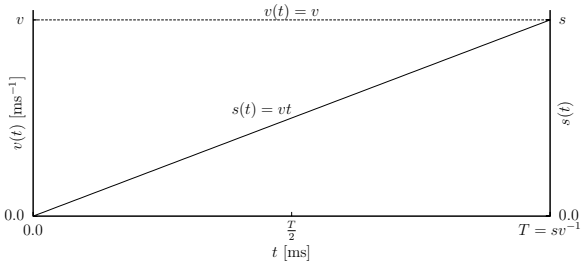
Pospešena interpolacijska funkcija (graf 3.8(c)) s podanim konstantnim pospeškom a linearno povečuje hitrost $v(t)$ ter kvadratično pot $s(t)$, dokler ta ne pride do končne poti s. Pot $s(t)$ je enako kot pri linearni interpolacijski funkciji zvezna C^0, saj ima $v(t)$ točko nezveznosti na koncu intervala $[0, T = \sqrt{2s/a}]$. Za ohranjanje zveznosti C^0 se pospešeni interpolacijski funkciji pri spremembi končne poti s ni potrebno posebej prilagajati.

Pospešena in zavirana interpolacijska funkcija

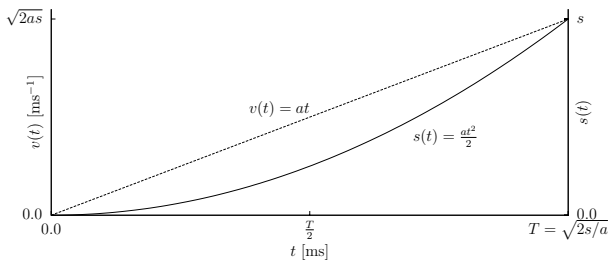
Ta interpolacijska funkcija (graf 3.8(d)) je podobna pospešeni interpolacijski funkciji, le da pospešuje samo prvo polovico časa, nato pa z nasprotno enakim pospeškom $-a$ zavira. Pot $s(t)$ je tu zvezna C^1, saj je prvi odvod $v(t)$ zvezen povsod, drugi odvod $a(t)$ pa ima tri točke nezveznosti (0, $T/2$ in T). V primeru spremembe končne poti s je prilagajanje te interpolacijske funkcije odvisno od tega, v katerem trenutku do te spremembe pride.



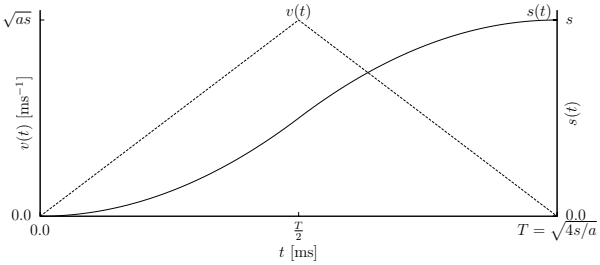
(a) Takojšna interpolacijska funkcija



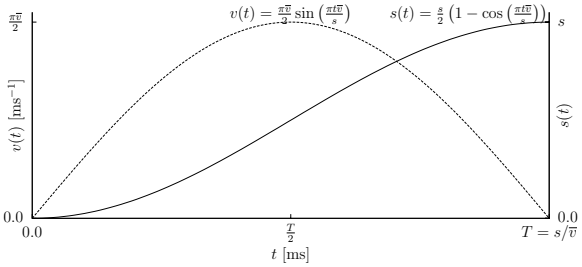
(b) Linearna interpolacijska funkcija



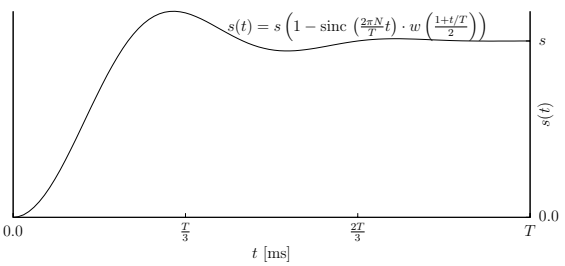
(c) Pospešena interpolacijska funkcija



(d) Pospešena in zavirana interpolacijska funkcija

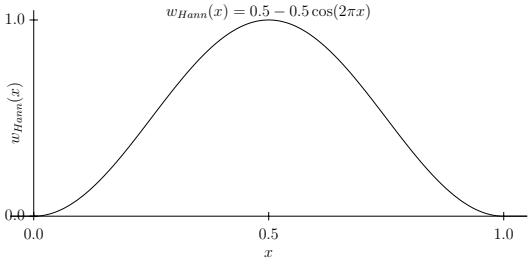


(e) Zglajena interpolacijska funkcija

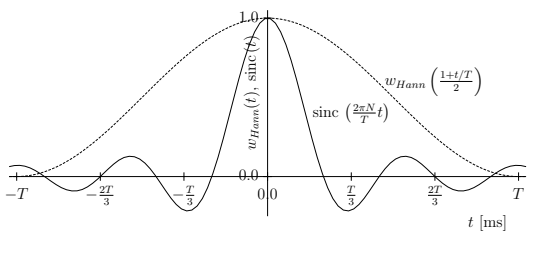


(f) Dušeno nihajoča interpolacijska funk. ($N = 1.5$)

Slika 3.7: Grafi hitrosti in poti interpolacijskih funkcij.



(a) Hannova okenska funkcija



(b) Funkcija sinc ($N = 1.5$) in okenska funkcija

Slika 3.8: Izpeljava dušeno nihajoče interpolacijske funkcije.

Zglajena interpolacijska funkcija

Zglajena interpolacijska funkcija (graf 3.8(e)) je zvezna C^1, četudi gre za zamaknjeno in raztegnjeno kosinusno funkcijo. Njen drugi odvod $a(t)$ ima namreč točki nezveznosti na robovih intervala $[0, T = s/\overline{v}]$. Zato pa je za razliko od pospešene in zavirane interpolacijske funkcije zvezen C^∞ povsod znotraj tega intervala. Za ohranjanje zveznosti C^0 je pri spremembni končne poti s kar precej dela, zato tega postopka tu ne bom opisoval.

Dušeno nihajoča interpolacijska funkcija

Dušeno nihajoča interpolacijska funkcija je v bistvu zamaknjena, raztegnjena in z okensko funkcijo porezana *sinc funkcija*. V primeru uporabe *Hannove okenske funkcije* (graf 3.9(a)) se nihanje sinc funkcije do časa T zvezno zaduši. Računa se tako, da se od 1 odšeje produkt zamaknjene Hannove funkcije in sinc funkcije, kot sta prikazani na grafu 3.9(b), ter se razliko pomnoži s končno potjo s. Pri številu valov $N = 1.5$ sinc funkcije je rezultat funkcija, prikazana na grafu 3.8(f). Spreminjanje končne poti s sredi prehajanja pri tej interpolacijski funkciji ni smiselno in zato v aplikaciji *jPresenter* ni podprto.

3.3.4 Ravninske interpolacijske funkcije

V poglavju 3.3.3 sem govoril o interpolacijskih funkcijah, ki slikajo čas v neko skalarno količino, ki smo ji rekli pot. Sedaj si bomo ogledali funkcije $\vec{r}(t)$, ki čas t preslikajo v točko na ravnini oziroma vektor dveh komponent (x, y).

$$\vec{r}(t) : [0, T] \rightarrow \Re \times \Re \;\; ; \;\; \vec{r}(0) = \vec{r_0}, \vec{r}(T) = \vec{r_1} \tag{3.5}$$

Potrebo po ravninskih interpolacijskih funkcijah sem izpostavil, ko je bilo govora o uporabniško vodenem prehajanju stanja oziroma o razredu `AdvancedRegulatedTransformTransition`. Ideja je, da neko prehajanje z zvezno hitrostjo sledi gibanju miške in s pomočjo ravninske interpolacijske funkcije oblaži sunkovite premike. Prehajanje se nato z neko časovno zakasnitvijo konča na mestu, kjer se je miška ustavila. Časovna zakasnitev in trajektorija, ki jo opiše, zavisita od lastnosti ravninske interpolacijske funkcije.

Ravninske interpolacijske funkcije izvajajo dve interpolaciji. Ena je interpolacija poti, ki preslika nek delež v koordinatno točko na trajektoriji. Druga interpolacija pa ugotavlja delež v odvisnosti od časa in gre v bistvu za eno izmed skalarnih interpolacijskih funkcij, opisanih v poglavju 3.3.3. Prva torej interpolira trajektorijo, druga pa gibanje po tej trajektoriji.

Linearna ravninska interpolacijska funkcija

Najpreprostejša izvedba ravninske interpolacijske funkcije je z uporabo linearne interpolacije, ki vmesno točko med $A = (A_x, A_y)$ in $B = (B_x, B_y)$ ob času t izračuna kot:

$$\left(A_x + (B_x - A_x) \frac{s(t) - s_A}{s_B - s_A}, A_y + (B_y - A_y) \frac{s(t) - s_A}{s_B - s_A} \right)$$

$s(t)$ je poljubna skalarna interpolacijska funkcija in s_B njena končna pot. s_A je skupna pot, opravljena do točke A. Končno pot s_B sproti prilagajamo kot skupno dolžino opravljene ter načrtovane trajektorije in je enaka vsoti evklidskih razdalj vseh parov zaporednih točk, ki trajektorijo opisujejo. Ker bi bilo vsakokratno preračunavanje celotne poti potratno in neučinkovito, lahko končno pot ob vsakem premiku miške popravimo le z izračunom njene spremembe. Če se na primer miška premakne iz točke B v B', ko je prehajanje v točki X med A in B, se nova končna pot izračuna po enačbi:

$$s_{B'} = s_B + \sqrt{\Delta_x(X,B')^2 + \Delta_y(X,B')^2} - \sqrt{\Delta_x(X,B)^2 + \Delta_y(X,B)^2}$$

Zglajena ravninska interpolacijska funkcija

Trajektorija, ki jo prehajanje opiše pri linearni ravninski interpolacijski funkciji, je C^0 zvezna, kar pomeni, da gre v bistvu za lomljeno črto. Sedaj bom opisal tri oblike zglajene ravninske interpolacijske funkcije, ki temeljijo na sestavljenih *Bézierovih krivuljah* tretje stopnje oziroma na posebnem primeru le-teh, imenovanem kubični *B-zlepek* (angl. *B-spline*) [14]. Sestavljene Bézierove krivulje so C^2 zvezne povsod razen v stičiščih dveh zaporednih krivulj, kjer jim bom zagotovil vsaj C^1 zveznost. Z B-zlepkom pa bom opisal celotno trajektorijo, kar pomeni, da bo tedaj interpolacijska funkcija C^2 zvezna vzdolž celotne svoje dolžine.

Parametrična definicija Bézierove krivulje tretje stopnje je za $t \in [0,T]$ podana z enačbo 3.6. V nadaljevanju bosta potrebna še njena prva odvoda (enačbi 3.7 in 3.8).

$$A(t) = \frac{1}{T^3}\left((T-t)^3A_0 + 3t(T-t)^2A_1 + 3t^2(T-t)A_2 + t^3A_3\right) \tag{3.6}$$

$$A'(t) = \frac{3}{T^3}\left((T-t)^2(A_1-A_0) + 2t(T-t)(A_2-A_1) + t^2(A_3-A_2)\right) \tag{3.7}$$

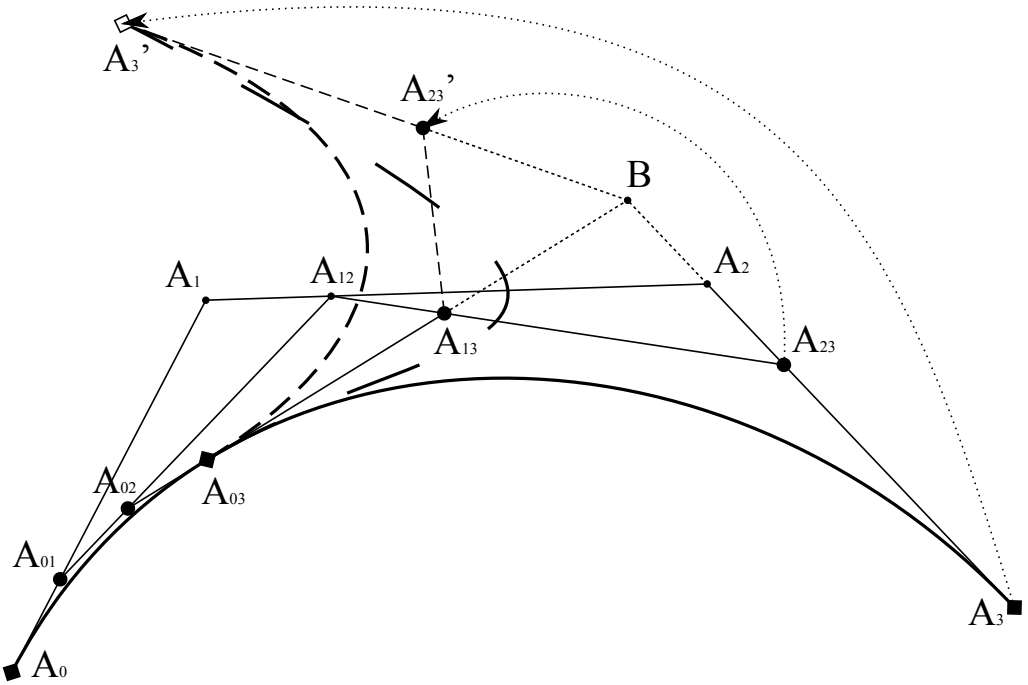
$$A''(t) = \frac{6}{T^3}\left((T-t)(A_0-2A_1+A_2) + t(A_1-2A_2+A_3)\right) \tag{3.8}$$

Sliki 3.9 in 3.10 z odebeljeno polno črto prikazujeta Bézierovo krivuljo s kontrolnimi točkami A_0, A_1, A_2 in A_3. Ko je prehajanje v času t in se nahajamo v točki A_{03}, se ciljna točka A_3 prestavi v A_3'. Točka A_{03} se na podlagi časa t ugotovi z uporabo *de Casteljaujevega algoritma*, tako da rekurzivno določimo točke $A_{(i)(j)} = A_{(i)(j-1)} \cdot \frac{1-t}{T} + A_{(i+1)(j)} \cdot \frac{t}{T}$, kjer izhajamo iz $A_{ii} = A_i$.

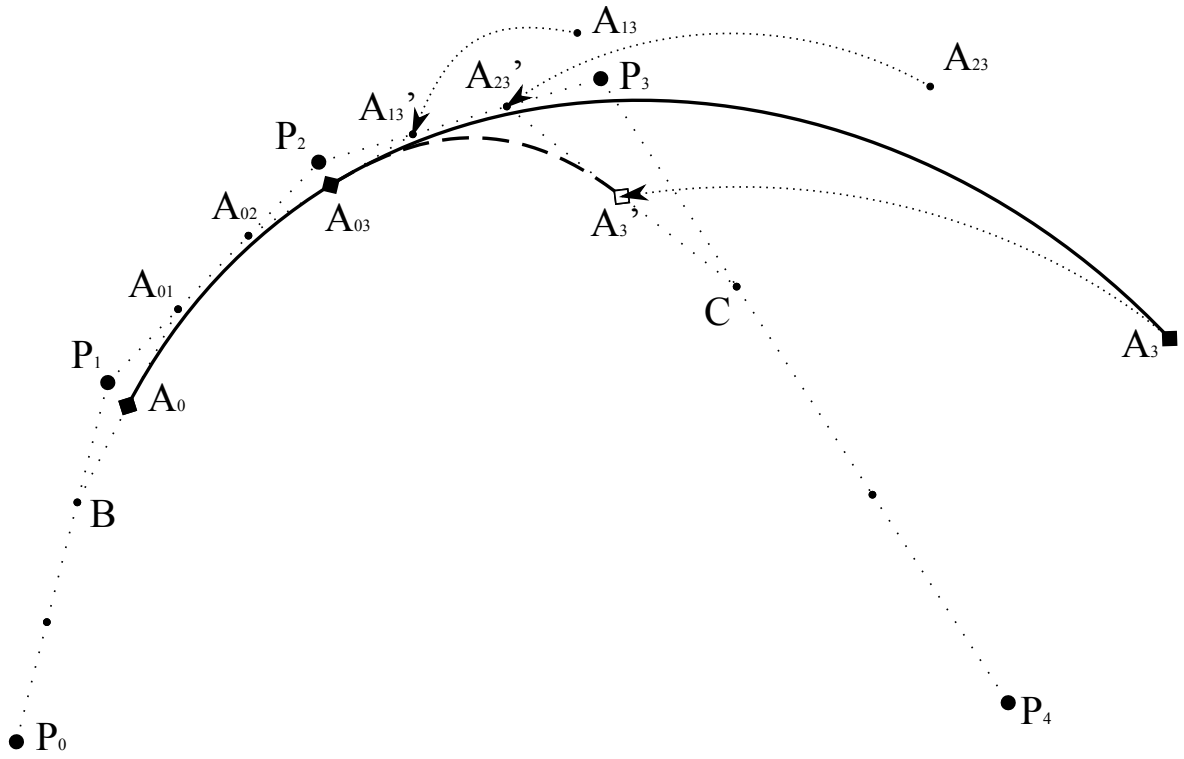
Omenjen algoritem v bistvu Bézierovo krivuljo $A(t)$ razbije na dve Bézierovi krivulji $A_a(t \in [0,T_a])$ in $A_b(t \in [0,T_b])$. $A_a(t)$ je opisana s kontrolnimi točkami A_0, A_{01}, A_{02} in A_{03}, $A_b(t)$ pa z A_{03}, A_{13}, A_{23} in A_3. Če želimo ohraniti C^1 zveznost v njunem stičišču A_{03}, mora veljati $A_a'(T_a) = A_b'(0)$. Po enačbi 3.7 to pomeni:

$$\frac{3}{T_a}(A_{03}-A_{02}) = \frac{3}{T_b}(A_{13}-A_{03}) \qquad \frac{A_{13}-A_{03}}{A_{03}-A_{02}} = \frac{T_b}{T_a} \tag{3.9}$$

Iz tega sledi, da je A_{13} enolično določena in da so A_{02}, A_{03} in A_{13} kolinearne.



Slika 3.9: Zglajena ravninska interpolacija po Bézierovi krivulji.



Slika 3.10: Zglajena ravninska interpolacija po B-zlepku.

Mimogrede naj omenim še, da so Bézierove krivulje že same po sebi po gibanju interpolirane tako, da je gibanje na bolj ravnih odsekih hitrejše od gibanja na krivinah. Četudi je to gibanje zvezno za $t \in (0, T)$, pa to ne velja za krajiščni točki $A(0)$ in $A(T)$. Torej nimamo speljevanja in zaviranja. Iz tega razloga je še vedno smiselna uporaba skalarne interpolacijske funkcije za preslikavo dejanskega časa v čas t, ki ga potrebujemo za izračun točke Bézierove krivulje.

Druga težava je določitev časa T za dano Bézierovo krivuljo. Ta čas bi lahko bil sorazmeren z dolžino krivulje, toda te dolžine ni mogoče analitično izračunati [22]. Za grob približek si lahko pomagamo z vsoto evklidskih razdalj zaporednih parov kontrolnih točk. Z uporabo rekurzivne poddelitve (angl. *recursive subdivision*) bi tak izračun konvergiral proti eksaktni rešitvi. Po drugi strani je smiselno v našem primeru uporabiti tudi konstanten in sorazmerno majhen T za vsak premik miške, ne glede na razdaljo. Tako bi zagotovili, da se prehod vedno konča v konstantnem času od zadnjega premika miške.

Oglejmo si sedaj tri možne rešitve prilagajanja trajektorije gibanja ob spremembi ciljne točke $A_3 \to A_3'$:

(i) V točki A_{03} (slika 3.9) ohranimo C^1 zveznost v skladu z enačbo 3.9. A_3' kot rečeno določi uporabnik z miško, za A_{13}' pa velja:

$$A_{13} = \frac{T_b}{T_a}(A_{03} - A_{02}) + A_{03} \tag{3.10}$$

Točka A_{23} je v tem primeru svobodna. Za funkcijo preslikave te točke želimo, da je reverzibilna. Torej, če uporabnik takoj prestavi miško nazaj v A_3, pričakujemo, da se tudi preslikava te točke preslika nazaj v A_{23}. Oglejmo si dva možna primera reverzibilne funkcije preslikave točke A_{23}:

(a) Točko A_{23} obdržimo na istem mestu. Na sliki 3.9 je dobljena krivulja označena z dolgo prekinjeno odebeljeno črto. Nadaljevanje krivulje je v tem primeru odvisno tudi od prvotno zastavljene poti (točke A_3).

(b) Točko A_{23} preslikamo v A_{23}' tako, da velja $\frac{A_3' - A23'}{A_{23}' - B} = \frac{A_3 - A23}{A_{23} - B}$, kjer je B presečišče premic, nosilk daljic $\overline{A_{03}A_{13}}$ in $\overline{A_{23}A_3}$. Na sliki 3.9 je dobljena krivulja označena s kratko prekinjeno odebeljeno črto. Za razliko od (a) se ta krivulja nadaljuje neodvisno od prvotno zastavljene poti.

(ii) V točki A_{03} (slika 3.10) želimo ohraniti C^2 zveznost. Veljati mora tako enačba 3.9 za zveznost v prvem odvodu kot tudi $A_a''(T_a) = A_b''(0)$. Z uporabo enačbe 3.8 slednjo enakost izpeljemo v:

$$\frac{6}{T_a^2}(A_{01} - 2A_{02} + A_{03}) = \frac{6}{T_b^2}(A_{03} - 2A_{13} + A_{23})$$

$$\frac{A_{03} - 2A_{13} + A_{23}}{A_{01} - 2A_{02} + A_{03}} = \left(\frac{T_b}{T_a}\right)^2 \tag{3.11}$$

$$A_{23} = \left(\frac{T_b}{T_a}\right)^2 (A_{01} - 2A_{02} + A_{03}) - A_{03} + 2A_{13} \tag{3.12}$$

Iz povedanega sledi, da sta v tem primeru enolično določeni tako točka A_{13} oziroma A'_{13} (enačba 3.10) kot tudi točka A_{23} oziroma A'_{23} (enačba 3.12). Poenostavljen primer take krivulje ($T_a = T_b$) je na sliki 3.10 prikazan s prekinjeno odebeljeno črto.

Krivuljo, sestavljeno iz n kubičnih Bézierovih krivulj, lahko opiše enakomeren (angl. *uniform*) B-zlepek z $n+3$ kontrolnimi oziroma *de Boorovimi* točkami (P_i, $i \in [0, n+2]$) [11]. To je štirimi za opis prve Bézierove krivulje in po eno za vsako nadaljnjo krivuljo, ki doda po eno svobodno točko.

Na sliki 3.10 naš primer krivulje opisuje B-zlepek s kontrolnimi točkami P_0, P_1, P_2, P_3 in P_4 na časovnem intervalu $[t_3, t_5]$. Za vozle $t_0 \ldots t_8$ smo rekli, da so enakomerni, veljati pa mora $t_5 - t_3 = T_a + T_b$. Čas točke A_{03} izračunamo po enačbi $t' = t_3 + (t_5 - t_3)\frac{T_a}{T_a+T_b}$ in je v našem primeru ($T_a = T_b$) kar enak $t' = t_4$.

Kontrolne točke lahko določimo z uporabo *de Boor-Coxovega algoritma* [9, 7], če izračunamo točke našega B-zlepka v trenutkih t_3, $t' = t_4$ in t_5. De Boor-Coxov algoritem je v bistvu posplošitev de Castaljaujevega algoritma za poljubne B-zlepke:

$$P(t) = P_i^k(t) \quad ; \; t \in [t_i, t_{i+1}] \tag{3.13}$$

$$P_i^j(t) = \begin{cases} (1 - r_i^j)P_{i-1}^{j-1}(t) - r_i^j P_i^{j-1}(t) & ; \; j > 0 \\ P_i & ; \; j = 0 \end{cases} \tag{3.14}$$

$$r_i^j = \frac{t - t_i}{t_{i+k+1-j} - t_i} \tag{3.15}$$

V našem primeru gre za kubičen enakomeren B-zlepek in je zato $k = 3$, za vozle pa lahko vzamemo $t_x = x$, saj se dolžine intervala pokrajšajo. Pri $t = t_3$ dobimo:

$$\begin{aligned}
P(t_3) = P_3^3(t_3) &= \left(1 - \frac{t_3 - t_3}{t_4 - t_3}\right) P_2^2(t_3) + \frac{t_3 - t_3}{t_4 - t_3} P_3^2(t_3) = P_2^2(t_3) \\
P_2^2(t_3) &= \left(1 - \frac{t_3 - t_2}{t_4 - t_2}\right) P_1^1(t_3) + \frac{t_3 - t_2}{t_4 - t_2} P_2^1(t_3) = \frac{1}{2}\left(P_1^1(t_3) + P_2^1(t_3)\right) \\
P_2^1(t_3) &= \left(1 - \frac{t_3 - t_2}{t_5 - t_2}\right) P_1^0(t_3) + \frac{t_3 - t_2}{t_5 - t_2} P_2^0(t_3) = \frac{2P_1 + P_2}{3} \\
P_1^1(t_3) &= \left(1 - \frac{t_3 - t_1}{t_4 - t_1}\right) P_0^0(t_3) + \frac{t_3 - t_1}{t_4 - t_1} P_1^0(t_3) = \frac{P_0 + 2P_1}{3}
\end{aligned}$$

Podobno lahko izračunamo še za $t = t'$ in $t = t_5$. Kontrolne točke določimo z upoštevanjem enakosti:

$$\begin{aligned}
P(t_3) &= P_2^2(t_3) = A_0 & P_2^1(t_3) &= A_{01} & P_1^1(t_3) &= B \\
P(t_4) &= P_3^2(t_4) = A_{03} & P_2^1(t_4) &= A_{02} & P_3^1(t_4) &= A'_{13} \\
P(t_5) &= P_4^3(t_5) = A'_3 & P_3^1(t_5) &= A'_{23} & P_4^1(t_5) &= C
\end{aligned}$$

3.3.5 Afine transformacije

Afine transformacije so transformacije, ki kolineranim točkam ohranjajo kolinearnost ter razmerja njihovih medsebojnih razdalj. Sestavljata jih linearna transformacija L in premik oziroma translacija $\vec{t}$:

$$\vec{x} \rightarrow L\vec{x} + \vec{t}$$

Linearne transformacije so med drugim lahko vrtenje oziroma rotacija, povečava (angl. *scale*), zrcaljenje (angl. *reflection*), striženje (angl. *shear*) in nenazadnje vse linearne kombinacije drugih linearnih transformacij. V aplikaciji *jPresenter* je podprta le kombinacija vrtenja R in premika $\vec{t}$. Z njo opisujemo bodisi prostorsko stanje nekega objekta ali pa transformacijo nad njim. Takim transformacijam bomo rekli rigidne transformacije:

$$\vec{x} \rightarrow R\vec{x} + \vec{t}$$

Opisane so z razredom `RigidTransform`, ki ga sestavljata informacija o položaju oziroma premiku, predstavljena z vektorjem treh komponent, ter informacija o usmerjenosti oziroma vrtenju, hranjena v obliki kvaterniona (dodatek A). Izbira take predstavitve je predvsem posledica potrebe po interpolaciji, ki je potrebna pri animaciji. Nad primerkom P omenjenega razreda lahko izvajamo naslednje operacije:

- Funkciji `set()` in `reset()` za nastavljanje transformacije.
- Funkcija `interpolate()` transformacijo primerka P za dan delež približa transformaciji drugega primerka, podanega kot parameter.
- Funkcija `postTransform()` trasformira P s transformacijo drugega primerka R ($P \leftarrow P \cdot R$).
- Funkcija `postRotate()` transformacijo primerka P obrne za dan kot ψ okrog poljubne osi $\vec{r}$ ($P \leftarrow P \cdot \mathrm{Rot}_{\vec{r}}(\psi)$). $\mathrm{Rot}_{\vec{r}}(\psi)$ je transformacija, ki opisuje tako rotacijo.
- Funkcija `postTranslate()` primerek P premakne za dan vektor $\vec{x}$ glede na orientacijo P ($P \leftarrow P \cdot \mathrm{Tr}_{\vec{x}}$). $\mathrm{Tr}_{\vec{x}}$ je transformacija, ki opisuje premik za $\vec{x}$.
- Funkcija `toMatrix()` pretvori transformacijo primerka P v polje decimalnih števil, ki predstavljajo transformacijsko matriko velikosti 4x4.
- Funkcija `toEuler()` pretvori usmerjenost primerka P v zaporedje treh *Eulerjevih kotov* pri vrtenju okrog osi koordinatnega sistema v podanem vrstnem redu.
- Funkcija `positionDist()` izračuna evklidsko razdaljo P do drugega primerka.
- Funkcija `orientationDist()` izračuna razliko med usmeritvijo primerka P ter drugega primerka, izraženo v kotu.

Vse transformacijske funkcije sprejemajo še parameter, ki pove, ali naj se transformira le položaj, usmeritev ali oboje.

3.4 Interakcija z uporabnikom

Najenostavnejša oblika interakcije, ki jo podpira aplikacija *jPresenter*, je sprejemanje ukazov za zamenjavo strani. Ukaze prožita levi in desni gumb na miški. V kombinaciji s tipkami *shift*, *ctrl* in *alt* lahko izberemo različne načine menjavanja predstavitvenih strani. Druga oblika interakcije pa je navigacija z držanjem levega gumba miške za vrtenje oziroma desnega gumba za premikanje kocke. Ob pritisku obeh gumbov hkrati lahko kocko zadržimo v poljubini nestabilni legi (stabilnost kocke sem razložil v poglavju 3.1). Obračanje koleščka ob držanju enega gumba na miški povzroča premikanje v Z smeri in ima učinek povečevanja oziroma pomanjševanja predstavitvene strani.

3.4.1 Zamenjava strani

Ukazi za zamenjavo strani kocko obrnejo okrog Y osi glede na osnovno lego za nek večkratnik pravega kota. Ob tem se popravijo tudi teksture ploskev, tako da prednja ploskev prikazuje trenutno, leva prejšnjo, desna pa naslednjo predstavitveno stran. Aplikacija *jPresenter* omogoča naslednja prehajanja za obračanje strani:

- *Takojšnja zamenjava strani* v bistvu niti ni prehajanje, temveč le v trenutku zamenja teksture posameznih ploskev.
- *Zglajeno obračanje kocke* povzroči, da se kocka po zglajeni interpolacijski funkciji $\texttt{funk}_G$ s hitrostjo v obrne za $\Delta\texttt{str}$ strani v desno. To prehajanje sproži naslednja programska koda:

  ```
  funk_G = new SmoothInterpolationFunction(v);
  scena.appendTransition(new ParallelTransition(
    new TransformStateTransition(kocka, stanje_P, funk_G, BOTH),
    new TransformTransition(kocka, stanje_D, Δstr, funk_G, ORIENTATION)
  ));
  ```

  Prehajanje je sestavljeno iz dveh delov, ki se izvajata vzporedno. Prvi del poskrbi, da se kocka postavi v stabilno lego, če slučajno še ni. Drugi del pa povzroči vrtenje za $\texttt{stanje}_D \cdot \Delta\texttt{str}$, kjer je $\texttt{stanje}_D$ v bistvu en obrat kocke za $-90°$ okrog Y osi.

  Pri tvorbi primerkov razredov `TransformStateTransition` in `TransformTransition` zadnji parameter določa, kateri del podanega prostorskega stanja (drugi parameter) naj se pri prehajanju odraža na objektu. Možne vrednosti so `ORIENTATION` za vrtenje, `POSITION` za premik in `BOTH` za oboje.

- *Dušeno nihajoče obračanje kocke* je enako zglajenemu obračanju kocke, le da uporabi dušeno nihajočo interpolacijsko funkcijo $\texttt{funk}_N$:

  ```
  funk_N = new SincInterpolationFunction(T, N, HannWindow.getInstance());
  scena.appendTransition(new ParallelTransition(
    new TransformStateTransition(kocka, stanje_P, funk_N, BOTH),
    new TransformTransition(kocka, stanje_D, Δstr, funk_N, ORIENTATION)
  );
  ```

- *Odmaknjeno dušeno nihajoče obračanje kocke* najprej kocko oddalji na mesto, ki ga določa stanje $\texttt{stanje}_O$, jo nato zavrti z dušeno nihajočo interplacijsko funkcijo ter vrne nazaj na začetno mesto. Zaporedno izvajanje prehajanj omogoča razred `SequentTransition`. Programska koda izgleda približno tako:

```
funkG = new SmoothInterpolationFunction(v);
funkN = new SincInterpolationFunction(T, N, HannWindow.getInstance());
scena.appendTransition(new SequentTransition(
  new TransformStateTransition(kocka, stanjeO, funkG, POSITION),
  new ParallelTransition(
    new TransformStateTransition(kocka, stanjeP, funkG, ORIENTATION),
    new TransformTransition(kocka, stanjeD, Δstr, funkN, ORIENTATION)
  ),
  new TransformStateTransition(kocka, stanjeP, funkG, POSITION)
);
```

3.4.2 Navigacija kocke z miško

Navigacijo kocke izvaja uporabniško vodeno prehajanje stanja, o katerem je bilo govora v poglavju 3.3.2 in je izvedeno v razredu `AdvancedRegulatedTransformTransition`. Informacijo o premikih in vrtenju mu posreduje primerek razreda `SWTCubeController`, ki jo pridobi s poslušanjem dogodkov, povezanih z miško na komponenti uporabniškega vmesnika `GLCanvas`. Ta razred v kombinaciji z razredom `CursorAutoHide` skrbi še za skrivanje miškinega kazalca po sekundi neaktivnosti v celozaslonskem načinu prikazovanja predstavitve.

Premiki miške na XY ravnini se lahko linearno preslikajo v premike kocke. Premik po Z osi, ki se odraža v povečavi, pa sem pojasnil že v poglavju 3.1.3. Nekoliko več matematike je potrebne pri vrtenju. V poglavju 3.1 sem omenil, da se kocka lahko vrti okrog X in Y osi glede na svojo trenutno osnovno lego ter da je vrtenje okrog X osi omejeno na interval $[-90°, 90°]$.

Vrtenje okrog X osi bo določal premik miške v navpični smeri, vrtenje okrog Y osi pa premik v vodoravni smeri. Pomemben je tudi vrstni red, v katerem se odrazita omenjeni vrtenji na kocko. Če bi se kocka najprej zavrtela za $-90°$ okrog Y osi, bi se X os preslikala v $X' = Z$. Posledično bi nato vrtenje okrog X' osi predstavitveno stran, obrnjeno proti kameri, sukalo okrog njene normale, kar pa v našem primeru ni zaželeno. Boljša rešitev je vrtenje najprej okrog X ter nato še okrog Y osi. Okrog normale bi se v tem primeru sukali le zgornja in spodnja ploskev, kar pa ni problematično, saj ti dve ploskvi vsaj zaenkrat ne služita ničemur.

Takemu načinu predstavitve vrtenja oziroma usmeritve objekta v prostoru rečemo *Eulerjevi koti*. Podani so s tremi vrednostmi kotov vrtenja okrog osi X, Y in Z v določenem vrstnem redu. Možnih vrstnih redov je 12, saj poleg vseh permitacij osi $(3!/0! = 6)$ lahko za tretjo os uporabimo še enkrat prvo $(3!/1! = 6)$, ki ima zaradi vrtenja okrog srednje osi tedaj (lahko) že drugo smer.

V našem primeru je najustreznejši vrstni red x-z-y. Za enolično določljive kote vrtenja ima namreč kot vrtenja okrog srednje osi razpon $[0, \pi]$, ostala dva pa $[0, 2\pi]$. Ožjo zalogo vrednosti si najlažje privščimo pri vrtenju okrog Z osi, saj je to že v specifikaciji problema fiksirano na 0. Slednja ugotovitev pride do izraza pri prevedbi kvaterniona, ki opisuje začetno usmeritev kocke, v Eulerjeve kote. Prevedbo izvaja razred `RigidTransform` (poglavje 3.3.5 in dodatek A, enačba A.22), potrebuje pa jo razred prehajanja `AdvancedRegulatedTransformTransition`. To prehajanje izniči oziroma povozi (poglavje 3.3.2) vsa prehajanja, ki se izvajajo pred njim. Pri izrisovanju novo usmeritev kocke izračuna kot $I \cdot \mathrm{Rot}_x \cdot \mathrm{Rot}_z \cdot \mathrm{Rot}_y$, kjer je Rot_z vedno identiteta.

Poglavje 4

Zaključek

Osrednja tema te diplomske naloge je uporaba 3-D grafike za prikaz multimedijskih predstavitev. Strojno pospešena 3-D grafika na tem področju danes sicer ni nekaj novega, je pa vsekakor še precejšnja redkost. Najbolj razširjeno orodje za delo s predstavitvami, *Microsoft PowerPoint*, jo koristi šele v najnovejši različici. Nekoliko dlje je prisotna le v programu *Apple Keynote*.

Posebnost aplikacije *jPresenter*, ki je bila razvita v sklopu tega diplomskega dela, je navidezna kocka, upodobljena z uporabo knjižnice *OpenGL*. Njena ploskev, ki je v začetnem stanju obrnjena proti kameri, prikazuje trenutno predstavitveno stran, prejšnja stran je na njeni levi ploskvi, naslednja pa na desni. Kocko lahko vrtimo, premikamo, približujemo in oddaljujemo. Pri zamenjavi strani se vsebina ploskev ustrezno popravi.

Poleg 3-D grafike je aplikacija zanimiva tudi, ker deluje v okolju *Java* in je načeloma prenosljiva na večino operacijskih sistemov. Deluje lahko samostojno ali pa kot razširitev razvojnega okolja *Eclipse IDE* ter drugih aplikacij, osnovanih na *Eclipse RCP*. Zasnovana je precej splošno in omogoča enostavno širjenje funkcionalnosti z dodajanjem novih vtičnikov.

Pri razvoju aplikacije je zanimiv izziv predstavljalo delo z animacijo in računanjem vmesnih stanj objektov (kocke) v prostoru (poglavje 3.3). Četudi bi bil konkreten problem najlažje rešljiv z uporabo Eulerjevih kotov, sem se odločil za izvedbo bolj splošnega modela predstavitve scene, ki usmeritve in vrtenja objektov obravnava s kvaternioni (dodatek A).

Pri navigaciji kocke sem obravnaval tudi problem blaženja naglih gibov miške. To sem dosegel s pomočjo glajenja trajektorije gibanja z Bézierovimi krivuljami ter B-zlepki (poglavje 3.3.4).

4.1 Težave in omejitve

Glede na zelo pestro ponudbo precej bolj celovitih programov za delo s predstavitvami je funkcionalnost aplikacije *jPresenter* za večino uporabnikov še preskromna. V trenutni izvedbi lahko prikazuje le statične predstavitvene strani, shranjene v obliki *PDF* dokumentov. Učinki, povezani z menjavo strani, so omejeni na obračanje navidezne kocke in

preprosto prehajanje oziroma zamenjavo tekstur ploskev. Prav tako ni mogoča priprava novih in spreminjanje obstoječih predstavitvenih dokumentov.

Strojne zahteve niso pretirano visoke, saj *OpenGL* zadovoljivo podpira že večina današnjih grafičnih kartic. Z vidika programske opreme pa mora uporabnik priskrbeti okolje *Java JRE* različice 1.5 ali novejše. Trenutno aplikacija preverjeno deluje vsaj v operacijskih sistemih *Microsoft Windows* in *Linux*. Izjema je le vtičnik za podporo *PDF* predstavitev na osnovi knjižnice *Poppler*, saj sem vmesnik *JNI* za delo s to knjižnico zaenkrat prevedel in povezal le za osmo različico distribucije *Fedora* operacijskega sistema *Linux*.

4.2 Ideje za nadaljnji razvoj

Možnosti za nadaljnji razvoj aplikacije je veliko. Nekaj sem jih nakazal že v poglavju 4.1, navedel pa bi lahko še:

- *Podpora drugih datotečnih zapisov za predstavitvene dokumente* (poglavje 1.3). Potrebna bi bila priprava ustreznih vtičnikov z razčlenjevalniki zapisov.
- *Možnost izbire privzetega načina prehajanja med sosednjimi stranmi za vsako stran posebej.* Nastavitve bi moral shranjevati vtičnik za podporo datotečnega zapisa dane predstavitve, na primer v datoteki z enakim imenom kot predstavitveni dokument, a drugačno končnico.
- *Kazalo in drugi podatki o predstavitvenem dokumentu na zgornji in spodnji ploskvi kocke.* Izbira povezave v kazalu bi kocko obrnila na izbrano stran.
- *Delovanje povezav v predstavitvenem dokumentu.* Izbira povezave bi obrnila kocko na izbrano stran oziroma povezavo odprla v ustrezni aplikaciji.
- *Dinamična vsebina posameznih predstavitvenih strani.* Predstavitvena stran tako ne bi bila več ploskev kocke, predstavljena kot končno vozlišče v grafu scene, temveč bi se naprej delila na posamezne gradnike strani. Vsak gradnik bi se nato lahko samostojno gibal, relativno glede na položaj strani v prostoru. Taka sprememba bi terjala tudi prilagoditev podatkovnih struktur, opisanih v poglavju 2.4.1.
- *Bolj realistično obnašanje kocke s simuliranjem fizikalnih pojavov.* Na primer, kocko bi lahko smatrali, kot da je vpeta na vzmet, ki jo vrača v stabilno lego, obenem pa je vložena v viskozen medij, ki duši nihanje. Za posamezno dinamično količino kocke x (usmeritev in položaj v prostoru) potem velja diferencialna enačba:

  $$\ddot{x} = -\omega^2(x - x_0) - \eta\dot{x} \ ,$$

  kjer je ω lastna frekvenca kocke, η opredeljuje dušenje, x_0 pa je stabilna vrednost količine. Tak pristop bi verjetno celo nekoliko poenostavil prilagajanje gibanja uporabnikovi interakciji, saj interakcija vpliva le na ciljno stabilno lego x_0 in potencialno trenutno hitrost $\dot{x}$, shema za integracijo diferencialne enačbe pa ni nič bolj zapletena.
- *Podpora za pripravo in urejanje predstavitvenih dokumentov.* Urejanje v tekstovnem (npr. LaTeX/*Beamer*, *ConTeXt* [13], *HTML*, *XML* ...) ali pa grafičnem načinu.
- *Deklarativno animirane in interaktivne predstavitve* (poglavje 1.2, [23]).

Dodatek A

Kvaternioni

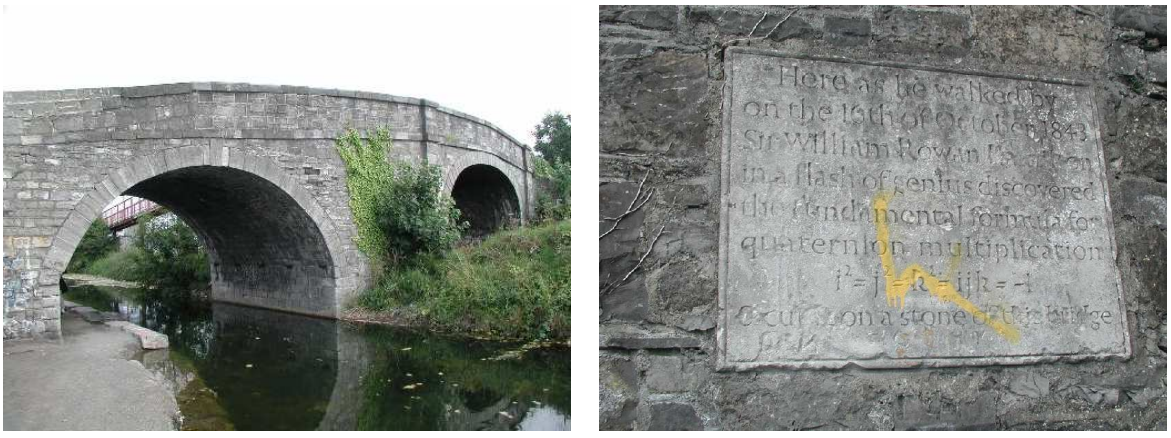
O afinih transformacijah je bilo govora že v poglavju 3.3.5. V tem dodatku se bom osredotočil na predstavitev usmeritev oziroma vrtenj v 3-D prostoru s kvaternioni [8].

Anekdota

Irski matematik William Rowan Hamilton je leta 1835 odkril, kako lahko kompleksna števila obravnavamo kot par realnih števil. Po spoznanju povezave med kompleksnimi števili in 2-dimenzionalno geometrijo si je naslednja leta prizadeval odkriti širšo algebro, ki bi imela podobno vlogo v 3-dimenzionalni geometriji.

Anekdota pravi [4], da je kvaternione odkril 16. oktobra 1843, med potjo na sestanek irske akademije. Nad odkritjem je bil tako navdušen, da je osnovno enačbo kvaternionske algebre A.1 vklesal v skalo Broughamskega mostu v Dublinu (slika A.1).

$$i^2 = j^2 = k^2 = ijk = -1 \tag{A.1}$$



Slika A.1: Broughamski most in spomenik v čast Williamu Rowanu Hamiltonu (sliki sta last Teviana Draya, vir [4]).

Definicija

Kasneje se je izkazalo, da je sprva iskal 3 namesto 4 dimenzionalno *normirano algebro z deljenjem* (angl. *normed division algebra*). Leta 1898 je Hurwitz [12] dokazal, da normirana algebra obstaja zgolj v dimenzijah $n = 1, 2, 4, 8$. Edine take algebre z multiplikativno identiteto so realna števila, kompleksna števila, kvaternioni in oktonioni [4, 12]. Normirana algebra je algebra z operacijo množenja $*$, za katero velja $||x * y|| = ||x|| * ||y||$, kjer $x, y \in \Re^n$.

Kvaternione se torej predstavi s štirimi realnimi števili $q_a = (a_1, a_2, a_3, a_4)$. Pogosto se uporablja tudi zapis z realnim številom in vektorjem treh komponent $q_a = (a_1, \vec{a})$, kjer je $\vec{a} = [a_2, a_3, a_4]$, ali pa zapis analogen zapisu kompleksnih števil (polarna oblika je podana z enačbo A.13):

$$q_a = a_1 \cdot 1 + a_2 \cdot i + a_3 \cdot j + a_4 \cdot k \tag{A.2}$$

V $\Re^4$ prostoru, se kvaternione lahko zapiše tudi v matrični obliki. Osnovni kvaternioni so matrično podani kot:

$$\begin{aligned} i &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad j = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix} \\ k &= \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix} \quad 1 = I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{A.3}$$

Za osnovne kvaternione veljajo tako imenovana Hamiltonova pravila:

$$\begin{aligned} i^2 = j^2 = k^2 &= -1 \\ ij = -ji &= k \\ jk = -kj &= i \\ ki = -ik &= j \end{aligned} \tag{A.4}$$

Operacije nad kvaternioni

- *Konjugacija kvaterniona*:

$$\overline{q_a} = \overline{a_1 \cdot 1 + a_2 \cdot i + a_3 \cdot j + a_4 \cdot k} = a_1 \cdot 1 - a_2 \cdot i - a_3 \cdot j - a_4 \cdot k \tag{A.5}$$

- *Norma kvaterniona*:

$$||q_a|| = \sqrt{q_a \overline{q_a}} = \sqrt{\overline{q_a} q_a} = \sqrt{a_1^2 + a_2^2 + a_3^2 + a_4^2} \tag{A.6}$$

Norma je multiplikativna $||q_a \cdot q_b|| = ||q_a|| \cdot ||q_b||$.

- *Normalizacija kvaterniona*:
$$\hat{q_a} = \frac{q_a}{||q_a||} \tag{A.7}$$
- *Seštevanje kvaternionov*:
$$q_a + q_b = (a_1 + b_1)1 + (a_2 + b_2)i + (a_3 + b_3)j + (a_4 + b_4)k \tag{A.8}$$
- *Množenje kvaternionov* se lahko izpelje iz Hamiltonovih pravil A.4:
$$\begin{aligned} q_a q_b = {} & (a_1b_1 - a_2b_2 - a_3b_3 - a_4b_4) + (a_1b_2 + a_2b_1 + a_3b_4 - a_4b_3)i + \\ & + (a_1b_3 + a_3b_1 - a_2b_4 + a_4b_2)j + (a_1b_4 + a_4b_1 + a_2b_3 - a_3b_2)k \end{aligned} \tag{A.9}$$
$$q_a q_b = (a_1, \vec{a})(b_1, \vec{b}) = (a_1b_1 - \vec{a} \cdot \vec{b}, a_1\vec{b} + b_1\vec{a} + \vec{a} \times \vec{b}) \tag{A.10}$$
Množenje ni komutativno ($q_a q_b \neq q_b q_a$), je pa asociativno ($q_a(q_b q_c) = (q_a q_b)q_c$).
- *Inverz kvaterniona*:
$$q_a^{-1} = \frac{\overline{q_a}}{q_a\overline{q_a}} = \frac{\overline{q_a}}{||q_a||^2} \tag{A.11}$$
Vkolikor je q_a normaliziran, je inverz kar enak $q_a^{-1} = \overline{q_a}$.
- *Interpretacija normaliziranega kvaterniona* q_a z vrtenjem v smeri urinega kazalca za Θ okrog enotskega vektorja $\hat{n_a}$:
$$q_a = (a_1, \vec{a}) = \left(\cos\left(\frac{\Theta}{2}\right), \hat{n_a}\sin\left(\frac{\Theta}{2}\right)\right) \tag{A.12}$$
Kvaternion q_a je normaliziran, saj velja $||q_a|| = 1$. Analogno polarnemu zapisu kompleksnih števil se kvaternion lahko zapiše tudi kot:
$$q_a = \cos\left(\frac{\Theta}{2}\right) + \hat{n_a}\sin\left(\frac{\Theta}{2}\right) = e^{\hat{n_a}\frac{\Theta}{2}} \tag{A.13}$$
- *Vrtenje točke oziroma vektorja* $\vec{p}$ v vektor $\vec{p'}$ za kvaternion q_a:
$$\vec{p'} = q_a\vec{p}q_a^{-1} \tag{A.14}$$
Množenje vektorja $\vec{p}$ s kvaternionoma q_a in q_a^{-1} je očitno, če se kvaterniona zapiše v obliki A.13, ali pa vektor $\vec{p}$ kot *nenormaliziran* kvaternion $(0, \vec{p})$ in je potem rezultat enak $(0, \vec{p'})$. Vektor p' se lahko sedaj zavrti še za kvaternion q_b v vektor p'':
$$\vec{p''} = q_b\vec{p'}q_b^{-1} = q_b(q_a\vec{p}q_a^{-1})q_b^{-1} = (q_bq_a)\vec{p}(q_a^{-1}q_b^{-1}) = q_bq_a\vec{p}(q_bq_a)^{-1} \tag{A.15}$$
Produkt kvaternionov $q_{a,b} = q_bq_a$ torej predstavlja vrtenje za q_a ter nato še za q_b.
- *k-kratno vrtenje vektorja* $\vec{p}$ za kvaternion q_a je enako enkratnemu vrtenju za kot $k\Theta$ okrog istega enotskega vektorja $\hat{n} = \hat{n_a} = \hat{n_b}$:
$$q_b\vec{p}q_b^{-1} = q_a^k\vec{p}q_a^{-k}$$
Sledi ugotovitev:
$$\left(\cos\left(\frac{\Theta}{2}\right), \hat{n}\sin\left(\frac{\Theta}{2}\right)\right)^k = q_a^k = q_b = \left(\cos\left(\frac{k\Theta}{2}\right), \hat{n}\sin\left(\frac{k\Theta}{2}\right)\right) \tag{A.16}$$

Sferična linearna interpolacija (*Slerp*)

Sferično linearno interpolacijo si lahko predstavljamo kot interpolacijo najkrajše poti oziroma loka po površini krogle z radijem 1, ki povezuje točki p_0 in p_1. Ω naj bo kot, ki ga oklepa ta lok glede na središče krogle. Torej velja $\cos\Omega = \vec{p_0} \cdot \vec{p_1}$ in $||\text{Slerp}\,(p_0, p_1; t)|| = 1$ za vsak $t \in [0, 1]$. S trigonometričnimi funkcijami se funkcija *Slerp* zapiše kot:

$$\text{Slerp}\,(p_0, p_1; t) = \frac{\sin(1-t)\Omega}{\sin\Omega} p_0 + \frac{\sin t\Omega}{\sin\Omega} p_1\ , \tag{A.17}$$

kjer je prehajanje ob času $t = 0$ v točki p_0 in ob času $t = 1$ v p_1 ter velja enakost $\text{Slerp}\,(p_0, p_1; t) = \text{Slerp}\,(p_1, p_0; 1-t)$.

Če se Ω približuje proti 0, lok med točkama p_0 in p_1 postaja vse bolj raven in *Slerp* postane linearna interpolacijska funkcija:

$$\text{Slerp}\,(p_0, p_1; t) = \lim_{\Omega\to 0}\left(\frac{\sin(1-t)\Omega}{\sin\Omega} p_0 + \frac{\sin t\Omega}{\sin\Omega} p_1\right) = (1-t)p_0 + tp_1 \tag{A.18}$$

Slerp se lahko računa tudi za interpolacijo vrtenja za kot Θ, če si množico vrtenj predstavljenih z normaliziranimi kvaternioni zamislimo kot polovico lupine enotske 4-dimenzionalne hiperkrogle (ta predstava sledi iz enačbe A.13). Polovico lupine zato, ker kvaterniona q in $-q$ predstavljata isto vrtenje. Zveza med kotom vrtenja Θ in kotom Ω, ki oklepa lok med kvaternionama na hiperkrogli, je določena z bijektivno preslikavo $\Theta = 2\Omega$.

Funkcijo *Slerp* je mogoče zapisati tudi v kvaternionski obliki. Pri interpolaciji normaliziranih kvaternionov iz q_0 v q_1 se najprej poišče kvaternion $q_d = (\cos\frac{\Theta}{2}, \hat{n_d}\sin\frac{\Theta}{2}))$, ki po najkrajši poti q_0 zavrti za kot Θ v q_1, torej velja $q_d q_0 = q_1$ (zaporedje množenj kveternionov je razloženo z enačbo A.15):

$$q_d = q_1 q_0^{-1}$$

Vmesno stanje v času t se izračuna tako, da se kvaternion q_0 namesto za q_d zavrti za $q_d^t = \left(\cos\frac{t\Theta}{2}, \hat{n_d}\sin\frac{t\Theta}{2}\right)$ (sledi iz zveze A.16):

$$\text{Slerp}\,(q_0, q_1; t) = (q_1 q_0^{-1})^t q_0 = (q_0 q_1^{-1})^{(1-t)} q_1 \tag{A.19}$$

Obravnava posebnih primerov:

(i) Ko je Ω oziroma Θ majhen, gre v bistvu za linearno interpolacijo po enačbi A.18.

(ii) Ko je $\Omega = \pi$, enačba A.17 ni izračunljiva, saj je pri iztegnjenem kotu neskončno možnih *najkrajših* interpolacijskih poti iz točke p_0 v p_1. Pri interpolaciji vrtenja za kot $\Theta = \pi$ s kvaternionom $q_d = (0, \hat{n_d})$ pa teh težav ni, saj gre za interpolacijo po $t\Omega \in [0, \frac{\pi}{2}]$ in je os vrtenja še vedno enolično določena z enačbo $q_d = q_1 q_0^{-1}$ (četudi teoretično obstajajo tudi druge osi z enako kratko interpolacijsko potjo). Posebna obravnava tega primera zato pri interpolaciji kvaternionov ni potrebna!

(iii) *Slerp* izvaja interpolacijo vrtenja po najbolj neposredni poti (loku krožnice) za kot $\Theta \in [-2\pi, 2\pi]$. Taki vrtenji sta seveda dve: krajše za $\leq 2\pi$ v eno smer in daljše za $\geq 2\pi$ v drugo smer. Pri izračunu kota Θ je potrebno torej paziti, da bo znotraj intervala $[-\pi, \pi]$, kar pa se da doseči z absolutno vrednostjo skalarnega produkta kvaternionov v enačbi A.20.

$$q_0 \cdot q_1 = ||q_0||||q_1|| \cos\frac{\Theta}{2} = \cos\frac{\Theta}{2} \qquad \Theta = 2\arccos|q_0 \cdot q_1| \tag{A.20}$$

Predznak skalarnega produkta je še vedno potrebno upoštevati pri smeri vrtenja.

Programska koda A.1 prikazuje izvedbo sferične linearne interpolacije kvaterniona q_0 v q_1 ob času t, ki enačbo A.19 izvede neposredno z izračunavanjem posameznih komponent interpoliranega kvaterniona po enačbi A.17. Zveza med enačbama A.19 in A.17 sledi iz računanja potence t po pravilu A.16.

```
1  public final Quaternion slerp(Quaternion q0, Quaternion q1, double t) {
2    double dot = q0.x*d1.x + q0.y*q1.y + q0.z*q1.z + q0.w*q1.w;
3    double s1, s2, Ω;
4    Quaternion q = new Quaternion();
5    if (dot < 0) {  // obravnavaj primer (iii)
6      dot = -dot;
7      s2 = -1.0;  // obrni smer vrtenja, ker smo vzeli absoluten 'dot'
8    } else {
9      s2 = 1.0;
10   }
11   if ((1.0 - dot) < ϵ) { // obravnavaj primer (i)
12     s1 = 1.0 - t;
13     s2 *= t;
14   } else {
15     Ω = arccos(dot); // Ω = Θ/2
16     s1 = sin((1.0-t)*Ω)/sin(Ω);
17     s2 *= sin(t*Ω)/sin(Ω);
18   }
19   q.w = s1*q0.w + s2*q1.w;
20   q.x = s1*q0.x + s2*q1.x;
21   q.y = s1*q0.y + s2*q1.y;
22   q.z = s1*q0.z + s2*q1.z;
23   return q;
24 }
```

Programska koda A.1: Izvedba sferične linearne interpolacije kvaternionov.

Pretvarjanje oblik

- *Normaliziran kvaternion $q = (w, x, y, z)$ v rotacijsko matriko $R = [r_{i,j}]$:*

$$R = \begin{bmatrix} w^2 + x^2 - y^2 - z^2 & 2xy - 2wz & 2xz + 2wy \\ 2xy + 2wz & w^2 - x^2 + y^2 - z^2 & 2yz - 2wx \\ 2xz - 2wy & 2yz + 2wx & w^2 - x^2 - y^2 + z^2 \end{bmatrix} \tag{A.21}$$

- *Rotacijsko matritko v normaliziran kvaternion [15]:* Rotacijska matrika je *ortogonalna matrika*, za katero velja, da je njena transponiranka enaka inverzu ($RR^T = R^T R = I$) in je *posebna*, kar pomeni, da je determinanta enaka 1 ($\det R = 1$). Slednja zahteva izključi zrcaljenja. Če gre za enostavna vrtenja okrog baznih vektorjev ($r_{1,2} = r_{2,3} = 0$, $r_{1,3} = r_{2,3} = 0$ ali $r_{1,2} = r_{1,3} = 0$), je kvaternione enostavno določiti po enačbi A.12, sicer pa se uporabi:

$$w = \pm\frac{1}{2}\sqrt{1 + r_{1,1} + r_{2,2} + r_{3,3}} \qquad S = \sqrt{r_{1,2}^2 r_{1,3}^2 + r_{1,2}^2 r_{2,3}^2 + r_{1,3}^2 r_{2,3}^2}$$

$$x = \begin{cases} \frac{r_{3,2} - r_{2,3}}{4w}, & w \neq 0 \\ \frac{r_{1,3} r_{1,2}}{S}, & w = 0 \end{cases} \quad y = \begin{cases} \frac{r_{1,3} - r_{3,1}}{4w}, & w \neq 0 \\ \frac{r_{1,2} r_{2,3}}{S}, & w = 0 \end{cases} \quad z = \begin{cases} \frac{r_{2,1} - r_{1,2}}{4w}, & w \neq 0 \\ \frac{r_{1,3} r_{2,3}}{S}, & w = 0 \end{cases}$$

- *Eulerjevi koti v normaliziran kvaternion:* Eulerjevi koti podani v x-y-z konvenciji, predstavljajo vrtenje najprej za kot ψ_x okrog vektorja $\vec{x} = (1, 0, 0)$, nato za ψ_y okrog $\vec{y} = (0, 1, 0)$ ter na koncu še za ψ_z okrog $\vec{z} = (1, 0, 0)$. Po enačbi A.12 se kvaternion $q_{x,y,z}$ izpelje kot produkt $q_z q_y q_x$ (kvaternioni se množijo v obratnem vrstnem redu!):

$$q_{x,y,z} = \begin{pmatrix} \cos(\psi_x/2)\cos(\psi_y/2)\cos(\psi_z/2) + \sin(\psi_x/2)\sin(\psi_y/2)\sin(\psi_z/2), \\ \sin(\psi_x/2)\cos(\psi_y/2)\cos(\psi_z/2) - \cos(\psi_x/2)\sin(\psi_y/2)\sin(\psi_z/2), \\ \cos(\psi_x/2)\sin(\psi_y/2)\cos(\psi_z/2) + \sin(\psi_x/2)\cos(\psi_y/2)\sin(\psi_z/2), \\ \cos(\psi_x/2)\cos(\psi_y/2)\sin(\psi_z/2) - \sin(\psi_x/2)\sin(\psi_y/2)\cos(\psi_z/2) \end{pmatrix}$$

- *Normaliziran kvaternion v Eulerjeve kote:* Pri izpeljavi enačb za neposredno pretvorbo normaliziranega kvaterniona v Eulerjeve kote, podane v x-y-z konvenciji, je potrebno najprej ugotoviti rotacijsko matriko kot produkt rotacijskih matrik vrtenja okrog baznih vektorjev $R = R_z R_y R_x$:

$$\begin{bmatrix} \cos\psi_y \cos\psi_z & \cos\psi_z \sin\psi_x \sin\psi_y - \cos\psi_x \sin\psi_z & \cos\psi_x \cos\psi_z \sin\psi_y + \sin\psi_x \sin\psi_z \\ \cos\psi_y \sin\psi_z & \cos\psi_x \cos\psi_z + \sin\psi_x \sin\psi_y \sin\psi_z & \cos\psi_x \sin\psi_y \sin\psi_z - \cos\psi_z \sin\psi_x \\ -\sin\psi_y & \cos\psi_y \sin\psi_x & \cos\psi_x \cos\psi_y \end{bmatrix}$$

  Iz enačenja prejšnje matrike ter matrike A.21 se izpeljejo zveze:

$$\frac{r_{2,1}}{r_{1,1}} = \frac{2xy + 2wz}{w^2 + x^2 - y^2 - z^2} = \tan\psi_z \qquad \frac{r_{3,2}}{r_{3,3}} = \frac{2yz + 2wx}{w^2 - x^2 - y^2 + z^2} = \tan\psi_x$$
$$r_{3,1} = 2wy - 2xz = \sin\psi_y$$

- *Pretvorba kvaternionov in Eulerjevih kotov po x-z-y konvenciji:* V poglavju 3.4.2 je bilo povedano, da aplikacija *jPresenter* potrebuje x-z-y konvencijo. Po tej konvenciji se Eulerjevi koti v kvaternione pretvorijo po enačbi:

$$q_{x,z,y} = \begin{pmatrix} \cos(\psi_x/2)\cos(\psi_y/2)\cos(\psi_z/2) - \sin(\psi_x/2)\sin(\psi_y/2)\sin(\psi_z/2), \\ \sin(\psi_x/2)\cos(\psi_y/2)\cos(\psi_z/2) + \cos(\psi_x/2)\sin(\psi_y/2)\sin(\psi_z/2), \\ \cos(\psi_x/2)\sin(\psi_y/2)\cos(\psi_z/2) + \sin(\psi_x/2)\cos(\psi_y/2)\sin(\psi_z/2), \\ \cos(\psi_x/2)\cos(\psi_y/2)\sin(\psi_z/2) - \sin(\psi_x/2)\sin(\psi_y/2)\cos(\psi_z/2) \end{pmatrix}$$

  Pretvorba kvaternionov v Eulerjeve kote po x-z-y konvenciji:

$$\begin{aligned} \psi_x &= \arctan\left(2(wx - yz), -x^2 + y^2 - z^2 + w^2\right) \\ \psi_z &= \arcsin\left(2(wz + xy)\right) \\ \psi_y &= \arctan\left(2(wy - xz), x^2 - y^2 - z^2 + w^2\right) \end{aligned} \tag{A.22}$$

  Pri enačbah A.22 velja omeniti še, da je potrebno posebej obravnavati primer, ko je kot ψ_z blizu $\pm\frac{\pi}{2}$. Takrat pride do tako imenovane kardanske zapore (angl. *gimbal lock*), saj se os Y preslika v prejšnjo lego osi X in se tako izgubi ena prostorska stopnja. Posledično enačbe A.22 takrat vračajo precej nestabilne rezultate.

Slike

Tabele

Algoritmi

Programska koda

Stvarno kazalo

Literatura

[1] DirectX. Microsoft. Dostopno na: http://www.microsoft.com/windows/directx/

[2] OpenGL overview. SGI. Dostopno na: http://www.opengl.org/about/overview/

[3] E. Angel, D. Shreiner, in V. Shreiner, "An interactive introduction to OpenGL programming," v *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*. New York, NY, ZDA: ACM, 2007, str. 1–124.

[4] J. C. Baez, "The octonions," 2001. Dostopno na: http://math.ucr.edu/home/baez/octonions/octonions.html

[5] W. Beaton in J. des Rivieres, "Eclipse platform technical overview," White Paper, The Eclipse Foundation, 2006. Dostopno na: http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf

[6] A. Blewitt. What is IAdaptable? Dostopno na: http://www.eclipsezone.com/articles/what-is-iadaptable/

[7] M. Cox, "The numerical evaluation of b-splines," *Journal of the Institute of Mathematics and its Applications*, zv. 10, str. 134–149, 1972.

[8] E. B. Dam, M. Koch, in M. Lillholm, "Quaternions, interpolation and animation," Department of Computer Science, University of Copenhagen, Kopenhagen, Danska, Tehn. por., 1998.

[9] C. De Boor, "On calculating with b-splines," *Journal of Approximation Theory*, zv. 6, str. 50–62, 1972.

[10] S. Delap. Understanding how Eclipse plug-ins work with OSGi. Dostopno na: http://www.ibm.com/developerworks/library/os-ecl-osgi/index.html

[11] E. Demidov. An interactive introduction to splines. Dostopno na: http://www.ibiblio.org/e-notes/Splines/Intro.htm

[12] H.-D. Ebbinghaus, R. Remmert, M. Koecher *in sod.*, "Numbers," ser. Graduate Texts in Mathematics. New York, NY, ZDA: Springer, 1988, pogl. Composition Algebras. Hurwitz's theorem – Vector-Product-Algebras, str. 265–280.

[13] H. Hagen. ConTeXt. PRAGMA Advanced Document Engineering. Dostopno na: http://www.pragma-ade.nl/

[14] K. I. Joy, "On-line geometric modeling notes." Dostopno na: http://www.css.taylor.edu/~btoll/s99/424/res/ucdavis/CAGDNotes/

[15] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006, chapter 4: The Configuration Space.

[16] T. Leech, *How to Prepare, Stage, and Deliver Winning Presentations*, 3. izd. New York, NY, ZDA: AMACOM, feb. 2004.

[17] OpenGL ARB, D. Shreiner, M. Woo, J. Neider, in T. Davis, *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, avg. 2005.

[18] "About the OSGi service platform," White Paper, OSGi Alliance, 2007. Dostopno na: http://www.osgi.org/documents/collateral/OSGiTechnicalWhitePaper.pdf

[19] I. Parker, "Absolute PowerPoint: Can a software package edit our toughts?" *The New Yorker*, str. 76–87, maj 2001.

[20] D. Reiners, "Scene graph rendering," ser. IEEE VR 2002 Course Notes. OpenSG Forum, mar. 2002.

[21] D. Reveman. Compiz. Dostopno na: http://en.opensuse.org/Compiz

[22] M. Walter in A. Fournier, "Approximate arc length parametrization," v *Proceedings of the 9th Brazilian Symposium on Computer Graphics and Image Processing*, Caxambu, Minas Gerais, Brazil, okt. 1996, str. 143–150.

[23] D. Zongker in D. Salesin, "On creating animated presentations," v *Proceedings of Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, 2003, str. 298–308. Dostopno na: http://citeseer.ist.psu.edu/zongker03creating.html

Izjava

Izjavljam, da sem diplomsko nalogo izdelal samostojno pod vodstvom mentorja prof. dr. Saše Divjaka. Izkazano pomoč drugih sodelavcev sem v celoti navedel v zahvali.

Ljubljana, 26. marec 2008 Anže Žagar